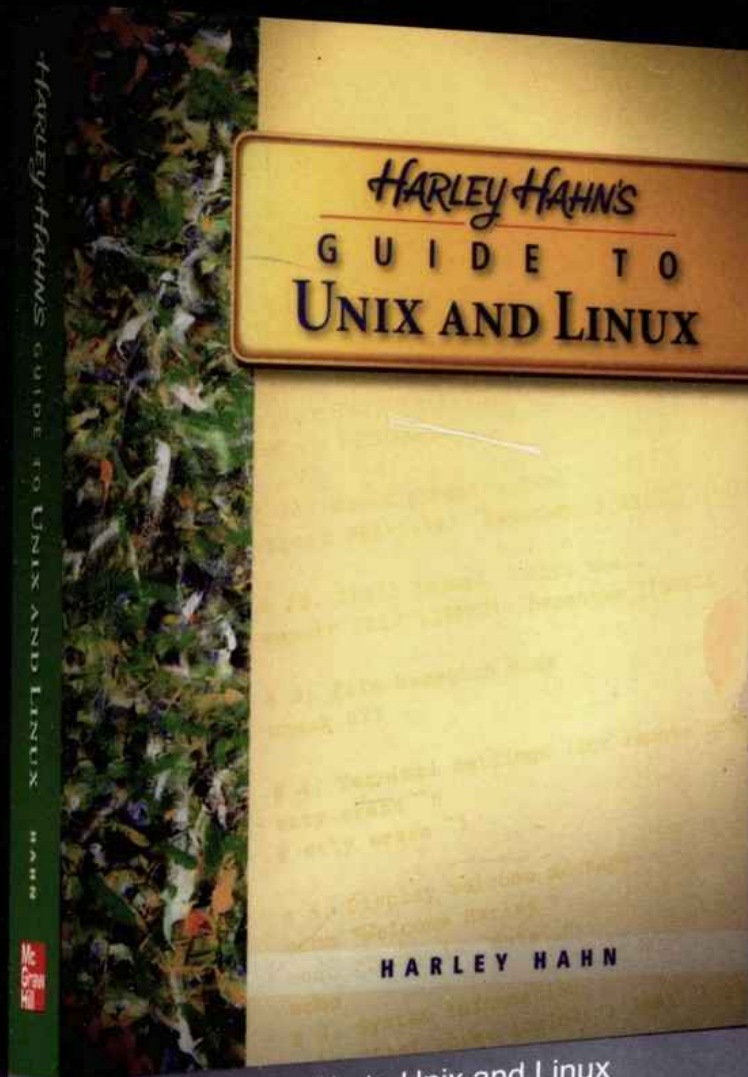


国外计算机科学经典教材

Unix & Linux 大学教程

(美) Harley Hahn 著 张杰良 译



Harley Hahn's Guide to Unix and Linux

McGraw Hill Education

清华大学出版社

- 面向对象分析与设计 (UML 2.0版)
- 数据结构与算法——C++版 (第3版)
- 数据挖掘原理与应用——SQL Server 2005 数据库
- 嵌入式系统——体系结构、编程与设计
- 信息技术教程 (第7版)
- 算法概论
- 计算机网络教程 (第4版)
- 工程学原理及问题求解 (第5版)
- 网络体系结构模式
- 数据库设计、应用开发和管理
- Linux管理基础教程 (第4版)
- 数据仓库工具箱
- 密码学与网络安全
- Java语言的科学与艺术
- C++类和数据结构
- Java大学教程 (第2版)
- 卓有成效的软件项目管理
- 操作系统 (第2版)
- 多媒体技术及应用 (第7版)
- 软件测试的有效方法 (第3版)
- 微处理器与外设大学教程 (第2版)
- Unix原理与应用 (原书第4版)
- 编程语言: 原理与范型 (第2版)
- 编译器设计

Unix是计算机发展历史上最成功的操作系统家族。它诞生于贝尔实验室,尔后迅速成为世界上操作系统的主流并延续至今。Linux派生于Unix,并且在小型机和桌面计算机领域成为和微软的Windows并驾齐驱的操作系统家族。Harley Hahn从20世纪80年代开始,亲身体验了几乎整个Unix以及Linux操作系统的发展历史。他所著的Unix和Linux教程,秉承了他的一贯风格,幽默风趣而又知识渊博。在本书中,不仅可以全面学习到Unix和Linux操作系统的工作原理和主要命令,还可以知晓Unix和Linux发展史上的许多奇闻趣事。

本书特色

- ◆ 解释了622个Unix专业术语,许多术语都给出了其历史来源
- ◆ 书中的示例涵盖了Linux、FreeBSD和Solaris等不同类型的操作系统
- ◆ 详细阐述了各种shell的异同并用示例加以说明

作者简介

Harley Hahn是一名多才多艺的作家和计算机专家,他总共撰著了32本书,总销量超过了两百万册。其中,Harley Hahn's Internet Yellow Pages一书是业界第一本销量超过一百万册的有关Internet的书。另外,他的Harley Hahn's Internet Insecurity和Harley Hahn's Internet Advisor还获得过普利策奖的提名。

信息网站: <http://www.tup.com.cn>
<http://www.tupwk.com.cn>
 读者信箱: wkservice@vip.163.com
 投稿邮箱: bookservice@263.net

Mc
Graw
Hill Education

<http://www.mheducation.com>

ISBN 978-7-302-20956-0



9 787302 209560 >

定价: 98.00元

TP316.8
H003

国外计算机科学经典教材

Unix & Linux大学教程

(美) Harley Hahn 著
张杰良 译

清华大学出版社
北 京

TP316.8
H003



Harley Hahn

Harley Hahn's Guide to Unix and Linux

EISBN: 978-0-07-313361-4

Copyright © 2009 by The McGraw-Hill Companies, Inc.

Original language published by The McGraw-Hill Companies, Inc. All Rights reserved. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Simplified Chinese translation edition is published and distributed exclusively by Tsinghua University Press under the authorization by McGraw-Hill Education(Asia) Co., within the territory of the People's Republic of China only (excluding Hong Kong, Macao SAR and Taiwan). Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书中文简体字翻译版由美国麦格劳-希尔教育出版(亚洲)公司授权清华大学出版社在中华人民共和国境内(不包括中国香港、澳门特别行政区和中国台湾地区)独家出版发行。未经许可之出口视为违反著作权法,将受法律之制裁。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

北京市版权局著作权合同登记号 图字:01-2009-2546

本书封面贴有 McGraw-Hill 公司防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

Unix & Linux 大学教程/(美)哈恩(Hahn, H.)著;张杰良译. —北京:清华大学出版社, 2010.1

(国外计算机科学经典教材)

书名原文: Harley Hahn's Guide to Unix and Linux

ISBN 978-7-302-20956-0

I. U… II. ①哈…②张… III. ①Unix 操作系统—高等学校—教材②Linux 操作系统—高等学校—教材
IV. TP316.8

中国版本图书馆 CIP 数据核字(2009)第 163759 号

责任编辑:王 军 李楷平

装帧设计:孔祥丰

责任校对:成凤进

责任印制:何 芊

出版发行:清华大学出版社

地 址:北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编:100084

社 总 机:010-62770175

邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者:清华大学印刷厂

装 订 者:三河市新茂装订有限公司

经 销:全国新华书店

开 本:185×260 印 张:52.25 字 数:1272 千字

版 次:2010 年 1 月第 1 版 印 次:2010 年 1 月第 1 次印刷

印 数:1~4000

定 价:98.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。联系电话:(010)62770177 转 3103 产品编号:030088-01

出版说明

近年来,我国的高等教育特别是计算机学科教育,进行了一系列大的调整 and 改革,亟需一批门类齐全、具有国际先进水平的计算机经典教材,以适应我国当前计算机科学的教學需要。通过使用国外优秀的计算机科学经典教材,可以了解并吸收国际先进的教学思想和教学方法,使我国的计算机科学教育能够跟上国际计算机教育发展的步伐,从而培养出更多具有国际水准的计算机专业人才,增强我国计算机产业的核心竞争力。为此,我们从国外多家知名的出版机构 Pearson、McGraw-Hill、John Wiley & Sons、Springer、Cengage Learning 等精选、引进了这套“国外计算机科学经典教材”。

作为世界级的图书出版机构, Pearson、McGraw-Hill、John Wiley & Sons、Springer、Cengage Learning 通过与世界级的计算机教育大师携手,每年都为全球的计算机高等教育奉献大量的优秀教材。清华大学出版社和这些世界知名的出版机构长期保持着紧密友好的合作关系,这次引进的“国外计算机科学经典教材”便全是出自上述这些出版机构。同时,为了组织该套教材的出版,我们在国内聘请了一批知名的专家和教授,成立了专门的教材编审委员会。

教材编审委员会的运作从教材的选题阶段即开始启动,各位委员根据国内外高等院校计算机及相关专业的现有课程体系,并结合各个专业的培养方向,从上述这些出版机构出版的计算机系列教材中精心挑选针对性强的题材,以保证该套教材的优秀性和领先性,避免出现“低质重复引进”或“高质消化不良”的现象。

为了保证出版质量,我们为这套教材配备了一批经验丰富的编辑、排版、校对人员,制定了更加严格的出版流程。本套教材的译者,全部由对应专业的高校教师或拥有相关经验的 IT 专家担任。每本教材的责编在翻译伊始,就定期不间断地与该书的译者进行交流与反馈。为了尽可能地保留与发扬教材原著的精华,在经过翻译、排版和传统的三审三校之后,我们还请编审委员或相关的专家教授对文稿进行审读,以最大程度地弥补和修正在前面一系列加工过程中对教材造成的误差和瑕疵。

由于时间紧迫和受全体制作人员自身能力所限,该套教材在出版过程中很可能还存在一些遗憾,欢迎广大师生来电来信批评指正。同时,也欢迎读者朋友积极向我们推荐各类优秀的国外计算机教材,共同为我国高等院校计算机教育事业贡献力量。

清华大学出版社

目 录

第 1 章 Unix 简介.....	1
1.1 使用 Unix 的原因.....	2
1.2 Unix 语言.....	3
1.3 Unix 学习提示.....	3
1.4 不知道正在使用 Unix 的人.....	4
1.5 知道正在使用 Unix 的人.....	4
1.6 从本书获取最多的知识.....	5
1.7 本书所做的假定.....	5
1.8 本书未做的假定.....	6
1.9 本书使用方法.....	6
1.10 练习.....	7
第 2 章 什么是 Linux? 什么是 Unix.....	8
2.1 什么是操作系统.....	8
2.2 什么是内核.....	9
2.3 Unix=内核+实用工具.....	11
2.4 “Unix”曾经是一个 专用名称.....	11
2.5 “Unix”现在是一个 通用名称.....	12
2.6 自由软件基金会.....	12
2.7 GNU 宣言摘录.....	14
2.8 GPL 和开放源代码软件.....	16
2.9 20 世纪 70 年代的 Unix: 由贝尔实验室转向 Berkeley.....	17
2.10 20 世纪 80 年代的 Unix: BSD 和 System V.....	18
2.11 1991 年的 Unix: 等待中.....	20
2.12 真命天子: Linus Torvalds.....	22
2.13 Linux 发行版.....	24
2.14 BSD 发行版.....	25
2.15 您应该使用什么类型的 Unix.....	26

2.16 获取 Linux 或者 FreeBSD 的方式.....	28
2.17 什么是 Unix? 什么是 Linux.....	31
2.18 练习.....	31
第 3 章 Unix 连接.....	33
3.1 人、机器和外星人.....	33
3.2 价格昂贵的早期计算机.....	34
3.3 主机和终端.....	36
3.4 终端室和终端服务器.....	38
3.5 控制台.....	40
3.6 Unix 连接.....	40
3.7 没有控制台的主机.....	42
3.8 客户端/服务器关系.....	43
3.9 按下键时发生的事情.....	44
3.10 字符终端和图形终端.....	46
3.11 最常见类型的终端.....	47
3.12 练习.....	48
第 4 章 开始使用 Unix.....	49
4.1 系统管理员.....	49
4.2 用户标识和口令.....	50
4.3 登录(开始使用 Unix).....	51
4.4 登录之后发生的事情.....	53
4.5 着手工作: shell 提示.....	54
4.6 注销(停止使用 Unix): logout、 exit、login.....	55
4.7 大写字母和小写字母.....	57
4.8 Unix 会话样本.....	57
4.9 改变口令: passwd.....	59
4.10 口令选择.....	60

Unix 简介

Unix 包括一系列在全世界广泛使用,并且可以在几乎所有类型的计算机上运行的操作系统。本书主要讨论 Unix 的使用,也讨论了 Linux, Linux 是 Unix 的一种。

我们将在第 2 章详细讨论 Unix。眼下,您需要知道的就是操作系统是一个运行计算机的总控制程序。

Unix 有许多类型,其中一些属于 Linux,一些不是。一般来讲,所有类型的 Unix 都非常相似,因此从实际操作来讲,只要您知道使用一种类型的 Unix,您就能使用所有类型的 Unix。

第一个 Unix 系统于 1969 年开发,开发者是 AT&T 公司贝尔实验室的一名程序员。该程序员开发该系统的目的是为了运行一个称为 Space Travel^{*}的游戏。现在,Unix 可以说是一个全球范围的文化,由许多工具、思想和习俗构成。

整体来讲,现代 Unix 系统非常庞大和复杂。实际上,没有一个人能够知道 Unix 的所有内容,即便是某一种特定类型的 Unix。甚至没有一个人能够知道 Unix 的大部分内容。

我意识到这可能看上去非常奇怪。毕竟,如果 Unix 如此复杂的话,它是如何出现的呢?谁来创建并增强 Unix 呢?还有,在出现问题时又是谁来改变 Unix 并修复问题呢?

在第 2 章中,当谈论 Unix 的历史以及目前 Unix 的维护方式时,我将回答这些问题。

眼下,我希望您能够理解的是 Unix 不仅仅是一种操作系统——Unix 简直就是一种文化。因此,在您阅读本书,思考您所学内容时,要意识到您不仅仅是在学习如何使用另一种计算机工具。您正在成为全球 Unix 社区中的一员,这是历史上全球最大的一个聪明人团体。

如果以前没有使用过 Unix,或许您会有一些激动惊奇。Unix 并不容易学习,但是它的设计非常出色,功能也特别强大,而且一旦您习惯了使用 Unix,就会发现极大的乐趣。

对于所有的计算机系统来说,解决方法不是那么明显的问题时有发生。在学习使用 Unix 的过程中,也会不时地遇到挫折或者失败。但是,不管发生了什么事,我可以保证一件事情:您永远不会对 Unix 感到厌烦。

^{*} Space Travel 游戏模拟太阳和行星的运动,以及一个可以在不同位置着陆的宇宙飞船。这里提到的程序员是 Ken Thompson。他和贝尔实验室的其他一些人一起开发了第一个成熟的 Unix 操作系统(大概是在他们玩累了 Space Travel 游戏之后)。

1.1 使用 Unix 的原因

您即将开始学习的 Unix 文化包含有大量需要学习和使用的工具。您可以创建并管理信息——文本文件、文档、图形、音乐、视频、数据库和电子表格等，方式多得超乎您的想象；您可以访问 Internet、浏览网页、发送电子邮件、传递文件以及参加各种讨论组；您可以玩游戏；您可以设计自己的网页，甚至还可以运行自己的 Web 服务器；您还可以使用许多不同的语言和编程工具编写计算机程序。

当然，您也可以使用其他操作系统完成所有这些事情，如 Windows，那么我们为什么还要学习 Unix 呢？

理由有许多方面，但是此时来看，这些理由都侧重于技术面，因此下面先介绍 4 个学习 Unix 的最重要的理由。

首先，使用 Unix，您可以决定如何使用计算机以及希望在细节上深入到何处程度。您可以按照自己的方式使用计算机，不用再按照他人(例如微软公司、IBM 公司或者您的母亲)设置的方式使用计算机。您可以根据系统是否适合自己来定制系统，而且还可以选择许多设计出色的工具和应用程序。

其次，使用 Unix 将改变您的思考方式，并且向好的方向转变。我相信如果您学习过如何阅读莎士比亚、聆听莫扎特或者欣赏梵高的绘画，那么在一定程度上，您将成为一名出色的人物。这一道理也适用于 Unix 的学习。

此时，如果您不相信上述观点，我也不会感到遗憾。但是在您阅读完本书之后，记着要回到本章，重新阅读本节内容，看看它是不是正确的。

第三，作为全球 Unix 社区的一名成员，您将有可能学习到如何使用一些目前人类所发明的最好工具。

另外，您可以数月连续使用一台计算机，而不必重启计算机。您不必担心计算机系统的崩溃、失去响应或者意外停止，而且——除非您管理一个大型的网络——您不必考虑下述令人恼火的事情：计算机病毒、间谍软件、运行失控的程序或者为了保持计算机平稳运行而必须执行的神秘、无法理解的规定程序。

最后，如果您是一名程序员(或者希望学习如何成为一名程序员)，那么您将发现一大批基于 Unix 的神奇工具可以用来帮助开发、测试及运行程序：拥有与语言相关的插件的文本编辑器、脚本解释器、编译器、交叉编译器(cross-compiler)、调试器、仿真器、语法分析程序生成器(parser generator)、GUI 构建器、软件配置管理器、错误跟踪软件、编译管理器(build manager)以及文档工具。另外，大多数类型的编程都拥有相应的社区以及各自的站点、电子邮件列表和讨论组，以及综合的软件文档。

当然，我不可能在本书中详细介绍 Unix 文化的各个方面。如果我试图这样做，那么您和我都将无所适从。因此，我选择一些基本的知识进行介绍。阅读完本书之后，您将理解最重要的概念，并能够使用最重要的工具。当遇到问题时，您还可以自己学习。为了开始学习 Unix，您只需要任何一台运行某种类型的 Unix(如 Linux)的计算机、Internet 连接，以及大量的时间和耐心。

另外还有一点：在大多数情况下，所有的东西——包括升级软件——都是免费的。

1.2 Unix 语言

环顾世界，Unix 系统的第一语言是美国英语。然而，Unix 系统和文档已经被翻译为许多其他语言，因此只要系统以自己的语言运行，就不必精通英语。但是，如果您进入世界范围的基于 Unix 的社区，就会发现大多数信息和许多讨论组都使用英语。

另外，Unix 社区还创造了它自己的许多新单词。在本书中，我将特别关注这样的单词。每次介绍新单词时，都将使用大写字母介绍。我将确保解释清楚单词的含义，并示范单词的使用方法。为了方便读者，书中所有术语的定义都集合到本书后面的术语表中了。当谈论某个具有特殊历史色彩的名称时，我将采用下述方式给予特殊解释。

名称含义

Unix

20 世纪 60 年代，贝尔实验室(属于 AT&T 公司)的一些研究人员在麻省理工学院开发一个称为 Multics 的项目，即一种早期的分时操作系统。Multics 是一个协作项目，程序员包括麻省理工学院、通用电气公司和贝尔实验室的人员。Multics 是“Multiplexed Information and Computing Service”的首字母缩写(Multiplex 指将多个电子信号组合成一个单独的信号)。

到 20 世纪 60 年代末，贝尔实验室的管理部门决定不再继续支持 Multics 项目，并将他们的研究人员撤回贝尔实验室。1969 年，这些研究人员中的一名研究员 Ken Thompson 为微型计算机 PDP-7 开发了一个简单的小型操作系统。在为该操作系统寻找名称时，Thompson 将他的系统和 Multics 进行了对比。

Multics 的目标是在同一时间为多个用户提供众多功能。Multics 非常庞大，难以使用，而且还有许多问题。

Thompson 的系统比较小、要求较低(至少在刚开始时)，而且一次只能由一人使用。另外，系统的每个部件只限于完成一件事情，并且出色地完成这件事情。Thompson 决定将他的系统命名为 Unics(“Uni”意味着“一个”)，后来很快又将 Unics 修改成 Unix。

换句话说，Unix 这个名称是 Multics 的双关语。

1.3 Unix 学习提示

在阅读本书时，您将注意到本书中提供了许多关于学习 Unix 的提示。这些提示是一些对于新手和有经验的用户都很重要的看法和捷径。为了强调这些提示，本书以下述特殊格式提供它们：

提示

Unix 是有趣的。

1.4 不知道正在使用 Unix 的人

什么类型的人使用 Unix 呢？

从字面上来讲，这个问题没有确定的答案。Unix 系统拥有遍布全球的使用者，因此它非常难以概括。但是，这并不能阻止我尝试回答这个问题。

广义上讲，我们可以将 Unix 用户分成两部分：一部分用户知道他们正在使用 Unix，另一部分用户则不知道他们正在使用 Unix。

大多数使用 Unix 的人不知道他们正在做什么。这是因为 Unix 可以在许多不同类型的计算机系统上使用，而且这些系统运行得非常出色，以至于使用它们的人意识不到他们正在使用 Unix 系统。

例如，大多数 Web 服务器运行在某种类型的 Unix 上。当访问网站时，人们通常意识不到使用的就是 Unix，至少无法直接意识到。Unix 还被许多商业机构、学校和组织使用。当您偶尔使用它们的计算机系统时，例如进行预订、查询信息、控制机器、注册班级或者办公时，您也有可能在不了解的情况下使用 Unix。

另外，Unix 可以用来运行所有类型的机器，不仅包括所有规模的计算机(从最大的大型机到最小的手持式设备)，而且还有嵌入式或实时系统，例如仪表、电缆调制解调器、移动电话、机器人、卡拉 OK 机器、收银机等。

最后，大多数支持 Internet 的机器也运行 Unix。例如，将数据从一点传递到另一点的计算机(路由器)都使用某种类型的 Unix，大多数的邮件服务器(存储电子邮件)和 Web 服务器(向外发送网页)也都使用 Unix。一旦理解了 Unix，您就会发现在使用网络的过程中遇到的许多特性就会讲得通。例如，您将会理解在输入网址时为什么必须注意字母的大小写(因为大多数 Web 服务器运行在某种类型的 Unix 系统上，正如所知，Unix 系统是区分大小写的)。

根据我的理解，不知道自己正在使用 Unix 的最有趣的一组人就是那些 Macintosh 用户。数百万人使用装有 OS X 系统的 Mac 机，然而他们不知道自己使用的操作系统就是 Unix，OS X 实际上基于一种称为 FreeBSD 的 Unix(我们将在第 2 章进一步讨论 OS X)。这就是 Mac 机器为什么如此可靠的原因之一，特别是与运行 Windows 系统(该系统显然不基于 Unix)的 PC 机相比时。

1.5 知道正在使用 Unix 的人

对于那些知道他们正在使用 Unix 的人，情况又怎么样呢？换句话说，什么类型的人愿意选择学习 Unix 呢？

依我的经验来看，此类人(比如您和我)拥有下述 4 个特征。

首先，Unix 人士比较聪明，大多数情况下，Unix 人士要比平常人聪明许多。

其次，Unix 人士既懒惰又勤奋。他们不喜欢整天忙于工作——这不需要任何理由。但是当他们遇到一个感兴趣的问题时，他们会不停地寻找解决方案。

第三，Unix 人士喜欢阅读。现实世界中，大多数人们难以集中 5 分钟的注意力去读书，

而 Unix 人士在遇到问题时竟然可以去阅读手册。

最后,当 Unix 人士使用计算机时,他们希望发挥主动作用,感觉是自己在控制计算机。他们不希望感觉是在被计算机控制。

您对自己感到疑虑吗?如果是的话,不用担心。您阅读本书并且已经阅读到这里这一事实已证明您有资格做一名 Unix 人士。

1.6 从本书获取最多的知识

我已经精心设计了此书,从而可以使您快速地查找到所需的内容。在开始之前,首先花一点时间查看一下本书的各个部分(我知道当我说这些话时,大多数人不愿意这样做,但是请大家无论如何也要查看一下本书的各个部分)。

首先,查看一下术语表。这可以让您从总体上了解一下本书将要讲授的重要概念。注意这里有许多内容需要学习。

其次,快速浏览一遍 vi 文本编辑器的“vi 命令小结”。一旦开始学习如何使用该程序(第 22 章),您将会发现这些索引非常有帮助。

除了术语表和“vi 命令小结”之外,在本书的后面还提供有两个 Unix 命令一览表。这两个列表为书中涉及的每条命令都提供了一行简要描述。

其中一个列表以字符顺序列举各条命令;另一个列表按命令分类列举命令。如果您希望做一些事情,但是不确定使用什么命令,那么这两个列表是查找命令的绝好位置。

如果希望查找特定主题的讨论,您可以在术语表中查找合适的术语。顺着对每个术语的定义看下去,就会找到详细解释该术语的参考章号。一旦知道了应该阅读哪一章内容,快速地浏览一下目录就可以明白应该阅读哪一节的内容。

1.7 本书所做的假定

在本书中,对于您所使用的 Unix 类型做了两个重要的假设。

第一,正如将在第 2 章中讨论的,Unix 有许多版本。现在,最流行的 Unix 系统是 Linux。大多数 Unix 系统包含相同的基本元素,因此在很大程度上,本书讨论的内容与使用的 Unix 类型无关。但是有时候,还是需要在 Linux 和非 Linux 功能之间进行选择。在这种情况下,我将倾向于选择 Linux,因为 Linux 是最流行的。

第二,正如第 4 章中所讨论的,读取并解释所输入命令的程序称为“shell”。在第 11 章中,我将向大家介绍可供选择使用的各种 shell。各种类型的 shell 之间没有什么实质性的区别。但是,在出现问题的少数几种情况中,我将使用一个名为“Bash”的特殊 shell。如果您希望使用另一个 shell,这也是可以的。如果使用了其他的 shell,那么除了少数几个细节不同外,不会有什么实际的问题。

1.8 本书未做的假定

如果您是一位有经验的计算机用户，希望学习 Unix 的知识，那么本书将是您的一本入门指南，在所有的重要领域向您提供坚实的背景知识。

但是，我没有假定您已经拥有一定的经验。如果您从来没有使用过计算机也没关系，您不需要知道 Unix。您不必是一名程序员，而且也无需具备有关电子学和数学方面的任何知识。

后面我将解释您需要知道的各个方面的内容。接下来请按您的节奏前进，继续阅读。

1.9 本书使用方法

在开始阅读和学习之前，应该意识到 Unix 世界充斥着巨量的信息，这一点很重要。为了开始学习 Unix，首先要阅读本书的前 7 章内容。这 7 章内容将引导您入门并教您一些基本的技能。

在适应了 Unix 之后，您知道了如何启动和停止工作会话、输入命令以及使用键盘，接下来就可以按任意的顺序阅读本书的其他章节。

提示

要想学习 Unix 的所有方面是不可能的。在学习 Unix 时要集中在您需要的以及感兴趣的内容上。

尽管我已经尽了最大的努力确保各章之间相互独立，但是您应该意识到各个主题之间还是互相依存的。没有什么最好的地方可以作为学习 Unix 的起始点，也没有一个最佳的顺序来学习 Unix 的各个方面。

例如，假设您希望定制自己的工作环境，那么，最好首先阅读 Unix 工作环境(第 6 章)。然后需要理解所谓的“shell”(第 11 章)，以及使用 shell 的一些细节内容(第 12、13 和 14 章)。此时，您就可以通过修改特定的文件来定制自己的工作环境了。

但是，为了修改文件，最好还要知道如何使用文本编辑程序(第 22 章)。因为您需要保存这些文件，所以还应该理解文件系统(第 23 章)、显示文件的命令(第 21 章)以及管理文件的命令(第 24 章和第 25 章)。当然，在输入信息之前，您需要理解如何启动一个工作会话(第 4 和 5 章)以及如何在 Unix 中使用键盘(第 7 章)。

很明显，这种学习顺序并不会快速地引您入门，但是它确实强调了在刚开始时就需要理解的最重要的原则：设计 Unix 的目的不是为了学习而是为了使用。换句话说，学习 Unix 既费力又费时。但是，一旦掌握了需要掌握的技能，无论用 Unix 从事什么样的工作都会既快又简便。

现在当您回过头来想想当初学习驾驶汽车时，您就会记得学习驾驶汽车根本不简单。但是一旦拥有了一些经验，您的动作就会熟练自然了。现在，您或许可以一手把着方向盘，边听音乐边与他人聊天。

我们将这一看法概括为下述提示。

提示

Unix 用起来容易，但学习起来难。

记住，一旦您阅读了本书前面几章的内容，就可以以任何顺序自学本书的其他主题。如果遇到一个您还无法理解的思想或者技能，那么您可以停下来看看另外一章的相关内容，或者跳过这一部分使您迷惑的内容，稍后再学。这就是人们在现实生活中学习 Unix 的方法：每次一点，取决于自己当时的需求。

一定不要去记忆每个细节。在某些章节中，我们讨论的问题有一定的深度。您可以学习一些自己感兴趣且有用的内容，而跳过其他内容。只要理解了基本的知识，并对可用的知识有一个基本印象，那么在需要时，您总可以返过头来再仔细学习。

提示

首先学习基本知识。然后再学习希望学习的内容，顺序可以由自己决定。

1.10 练习

1. 复习题

1. 第一个 Unix 系统是什么时候开发的？开发工作是在什么地方完成的？
2. 学习 Unix 的 4 个重要原因是什么？
3. “Unix”的最初名称是什么？

2. 思考题

1. 2001 年前，苹果公司 Macintosh 桌面计算机使用的操作系统是完全专有的。在 2001 年，苹果公司采用了一种基于 Unix 的新操作系统(OS X)。转向基于 Unix 的操作系统有哪 3 方面的优点？又有哪 3 方面的缺点？



什么是 Linux? 什么是 Unix

什么是 Unix? 什么是 Linux? 最简短的答案就是 Unix 是一种类型的计算机系统, 而 Linux 是 Unix 系统一个特定家族的名称。

如果您的奶奶突然问您“正在学习的 Unix 是什么东西啊?”, 那么上述答案已经足够了。但是, 在实践中, 您需要知道更多的内容。

在本章中, 我将全面解释 Unix 中最重要的思想, 特别是 Linux 中的重要理念。这样做的原因有两方面。首先, 我认为您会发现该讨论非常有趣。对我来说, Unix 系统是目前所开发的最神奇的计算机系统, 而且我希望分享这种感受。

其次, 一旦您对 Unix 有了基本的了解, 将会对我在本书中所讲授内容的背景有一个了解。通过这种方式, 当我们深入学习各种技术细节时, 就可以方便地记住细节内容。

在本章末尾, 将给出 Unix 和 Linux 的一个令人满意、便于使用的定义。但是, 在给出定义之前, 我不得不先解释一些重要的概念。为了打好基础, 我从所有计算机系统的一个最基本组件的讨论开始, 即操作系统。

2.1 什么是操作系统

计算机按照指令自动执行任务。一系列指令称为**程序**。因为计算机遵循指令, 所以我们称计算机在**运行**或者**执行程序**。一般而言, 程序被称为**软件**, 而计算机的物理部件被称为**硬件**。计算机硬件包括系统主板、磁盘驱动器、键盘、鼠标、显示器、屏幕、打印机等。

操作系统(属于软件)是运行计算机的总控制程序。操作系统的主要功能是高效地利用硬件。为了完成这一任务, 操作系统充当硬件的基本接口, 既为使用计算机的用户提供界面, 也为正在执行的程序提供界面。

无论何时, 当计算机启动并运行时, 操作系统就存在, 等待提供服务, 并管理计算机的资源。

例如, 假设您键入了一条显示文件名称的命令。在处理过程中, 正是操作系统来处理文件名称的查找细节以及在计算机屏幕上的显示。当运行一个需要打开一个新文件的程序时, 也正是操作系统来为文件预留存储空间并处理所有的细节。

更精确地说, 操作系统最重要的功能包括:

- 控制计算机并在计算机启动或者重新启动时初始化计算机。初始化过程只是引导过程的一部分。
- 支持与计算机交互所使用的界面(文本或者图形)。
- 为需要使用计算机资源(磁盘空间、文件位置、处理时间、内存等)的程序提供接口。
- 管理计算机的内存。
- 维护并管理文件系统。
- 调度工作。
- 提供账户和安全服务。

另外,所有的操作系统都打包有大量的程序以供使用。例如,有帮助创建、修改及管理文件的程序;有管理工作环境的程序;有与他人通信的程序;有访问网络的程序;有编写程序的程序等。Unix 提供了超过 1000 个这样的程序,每个程序都是一个执行特定作业的工具。

作为一个家族,所有 Unix 操作系统都有两个重要的特征:多任务和多用户。多任务意味着 Unix 系统可以同时运行不止一个程序。多用户意味着 Unix 可以同时支持不止一个用户(顺便说一下,微软的 Windows 系统是一个多任务、单用户的操作系统)。

名称含义

引导

术语“引导(booting)”是 bootstrapping 的简写,表示一个古老的谚语“通过自力更生出人头地”。例如,“在 Bartholomew 失去了所有的金钱之后,他好长一段时间非常贫困。但是,通过勤奋工作,经过自力更生他又出人头地了,成为了一个成功的细燕麦粉蛋糕商人。”

引导的思想就是一个困难的、复杂的目标可以通过一个小的动作开始,然后以这个小的动作为基础,一步一步地到达期望目标而完成。

计算机系统就是以这种方式启动的。当打开计算机的电源(或者重新启动计算机)时,一个单独的、小型的程序自动运行。这个程序启动另一个程序,一个更复杂的程序,然后逐步递进。最终,操作系统(一个非常复杂的程序)接过控制,完成初始化过程。

2.2 什么是内核

当计算机启动时,计算机要经历一系列的动作,这些动作构成引导过程。该过程的最后一个动作是启动一个非常复杂的程序,这个程序称为**内核(kernel)**。

内核的作用是控制计算机,充当操作系统的核心。由于这一点,所以内核总是一直运行。实际上,除非关闭了计算机系统,否则内核会一直运行。通过这种方式,内核一直可用,并在需要时提供基本的服务。

内核是操作系统的核心,它非常重要,因此我准备花一点时间来详细介绍内核。我们讨论的问题将有一定的技术深度,因此如果您觉得迷惑,千万不要自作聪明地点头来假装理解了。

尽管内核的本质可能根据操作系统的不同而有所区别,但是内核所提供的基本服务,

在各个操作系统之间都基本相同，这些服务包括：

- 内存管理(虚拟内存管理，包括分页)
- 进程管理(进程创建、终止、调度)
- 进程间通信(本地、网络)
- 输入/输出(通过设备驱动程序，即实现与物理设备实际通信的程序)
- 文件管理
- 安全和访问控制
- 网络访问(如 TCP/IP)

如果您对这些技术术语的含义有所了解，那再好不过了。如果还没有听说过这些术语，那也不必担心。只有极少数程序员才会实际考虑内核的这些内部细节。对于像您和我这样的人来说，最关键的事情就是知道内核是操作系统最重要的部分。实际上，我们稍后将会明白，Linux 和其他类型 Unix 之间的主要区别就是 Linux 使用了一个特殊的内核，该内核与其他 Unix 内核都有所不同。

内核有许多种类型，但是它们基本上可以分为两大类，规模较大的一类称为单内核(monolithic kernel)，规模较小的一类称为微内核(microkernel)。

单内核由一个非常庞大的程序构成，该程序自身可以完成所有的事情。微内核是一个非常小的程序，只能执行最基本的任务。为了执行其他功能，微内核要调用其他程序，这些程序称为服务器(server)。

单内核的优点是它的速度比较快：所有的事情都在一个单独的程序中完成，这样将比较高效。但是单内核的缺点是规模较大而且使用不便，从而使这类内核难以设计和维护。

微内核比较慢，这是因为它必须调用服务器来完成它的大部分工作，这样效率就不高。但是，因为采用了模块化设计，所以微内核易于程序员的理解，而且针对新系统修改微内核也比较快。微内核还有一个优点，即相比于单内核，它们更易于定制。

例如，假设您要为移动电话或者机器人创建一个操作系统。存储空间非常珍贵，因此一个小规模的内核要比大规模的内核好。另外，对于特殊目的的设备来说，可能不需要单内核的全部功能。在这种情况下，一个微内核加上经过精挑细选的服务器就是最佳的选择。

在创建新的操作系统时，设计人员可以从中进行选择。他们可以使用一个大型的单内核，也可以使用一个小型的拥有最少服务器的微内核。大多数 Unix 系统使用某种类型的单内核。但是，正如我们即将看到的，一些 Unix(如 Macintosh Unix，称为 OS X)使用的是微内核。

名称含义

内核

想象一个阿月浑子树坚果。坚果的外层是一个硬壳。坚果的里面是柔软、可食用的种子，从生物学上讲，称里面这部分为果仁(kernel)。因此，如果希望更加准确，就可以这样说，当我们吃阿月浑子树坚果时，我们敲破果壳，吃里面的果仁。

如果我们将 Unix 看作一个坚果，那么里面的就是果仁(kernel)，外面的就是壳。实际上，情况也确实如此。

刚才讨论的内核，就是所谓的操作系统核心。shell(第 11 章中讨论)就是“围绕”在内核周围的一种特殊类型程序(命令处理器)的名称，它充当我们与系统交互的个人界面。

2.3 Unix=内核+实用工具

前面已经解释过内核是操作系统的中心部分。内核会一直运行，它的工作就是执行基本的任务。

但是内核之外的其他内容有什么用处呢？

对于 Unix 来说，“内核之外的其他内容”包含大量的辅助程序，这些程序包含在 Unix 包中，可以分成若干种不同的类别。

其中最重要的程序是那些为用户提供使用计算机的界面的程序。这些程序是 shell 和 GUI。shell 是一种提供基于文本的界面的程序：您可以一个接一个地键入命令，shell 读取命令，然后完成所需的工作来执行命令。GUI(graphical user interface, 图形用户界面)是一个更复杂精美的程序，使用窗口、鼠标指针、图标等提供图形界面。我们将在第 5 章和第 6 章中讨论 shell 和 GUI，并且在第 11 章中详细阐述 shell，眼下，只需知道 Unix 中存在它们即可。

其他程序称为 Unix 实用工具，这些工具有数百个。每个实用工具(也就是每个程序)都是一个单独的工具。所有的 Unix 系统都提供有数百个这样的工具，它们是操作系统的一部分。其中一些实用工具是为程序员准备的，但是大多数实用工具对所有人都有用，即使是临时用户。另外，正如我们即将在第 15 章中讨论的，Unix 提供了综合利用现有工具解决新问题的方法。大量的工具，作为每个 Unix 系统的构成部分，是 Unix 功能如此强大的主要原因。

Unix 实用工具在每个 Unix 系统上的工作都相当一致。这意味着，在很大程度上，如果您知道使用一种类型的 Unix，那么您就知道所有实用工具的使用。实际上，本书的大部分内容致力于讲授如何使用最重要的 Unix 实用工具。我的目标是，在阅读完本书后，您将熟悉 Unix 中最重要的工具，这意味着您将一天比一天更熟悉使用 Unix。

那么，什么是 Unix 呢？

一个令人满意的答案就是 Unix 是一种类型的操作系统，它使用 Unix 内核，并且提供有许多 Unix 实用工具以及一个 Unix shell。实际上，大多数(但并不是全部)Unix 都提供有 shell 以及至少一个 GUI。

非正式地讲时，我们通常使用术语“实用工具”来包含 shell，因此，我们现在可以将 Unix 定义为：

Unix=Unix 内核+Unix 实用工具

2.4 “Unix”曾经是一个专用名称

在第 1 章中，我讲过第一个原始的 Unix 系统由程序员 Ken Thompson 在 1969 年开发。该工作是在贝尔实验室完成的，该实验室是 AT&T 公司的一个研究机构。从那时起，有无数人投身于 Unix 系统的开发，从而使其成为一个现代的操作系统家族。但是谁拥有实际名称“Unix”呢？

多年以来, Unix 由 AT&T 公司拥有, AT&T 公司坚持 Unix 必须总是以大写字母拼写, 即 UNIX。更准确地说, AT&T 公司的律师指出“商标 UNIX 必须总是以确切的印刷形式出现。”

另外, AT&T 公司的律师还宣布不能单独使用 UNIX 本身, 必须使用“UNIX 操作系统”。“商标 UNIX 不能用作名词, 必须总是用作形容词来修饰一般名词, 就如‘UNIX operating system’中一样。”

多少年来, 贝尔实验室一直是 Unix 研发的中心之一, 这一局势一直持续。1990 年, AT&T 公司组建了一个新的部门接管了 Unix。新的部门称为 Unix 系统实验室(Unix System Laboratory, USL)。1993 年 6 月, AT&T 公司将 USL 出售给 Novell 公司。1993 年 10 月, Novell 公司将名称“UNIX”的权利转移给一个叫 X/Open 的国际标准组织。最后, 1996 年, X/Open 和开放软件基金会(一个计算机公司协会)合并组成开放组织(The Open Group)。因此, UNIX 商标现在由开放组织拥有。

那么开放组织又是如何宣称 UNIX 的含义呢? 他们说“UNIX”是一个商标, 指任何通过他们认证, 并符合他们所谓的“单一 UNIX 规范”的操作系统。

2.5 “Unix”现在是一个通用名称

尽管开放组织(以及 AT&T 公司、USL、Novell 公司和 X/Open)声称拥有“UNIX”的权利, 但是多年以来, 单词“Unix”一直非正式地使用, 指任何类 Unix 的操作系统。从这意义上讲, Unix 有许多种不同的类型, 而且每年还有许多类型的 Unix 开发出来。

但是, 所有这些又提出了一个问题, 即“类 Unix”指什么呢?

答案有两种, 但是两种答案都不完全准确。第一种答案是如果操作系统既包含一个 Unix 内核以及一些 Unix 实用工具, 又可以运行能够在其他 Unix 操作系统上运行的程序, 那么这个操作系统就是 Unix 系统。

第二种答案是如果理解 Unix 的人说这个系统是 Unix, 那么它就是 Unix。

我意识到对于纯粹主义者——如苏格拉底或者信奉早期基督教教义的拉比——来说, 这样的非正式定义并不能令人满意。但是, 在现实世界中(当不在学校时, 我们生活的地方), 通常并不能总是提供完全准确的定义。

因此, 现在告诉您真相应该不会受到批评了: 如果想从技术上理解 Unix 是什么, 那么我无法告诉您 Unix 是什么——可能没有一个人能够做到——但是当我看到它时, 我知道它就是 Unix(而且, 有一天您也可以这样)。

2.6 自由软件基金会

现在您已经知道什么是 Unix 了, 下面开始回答下一个问题: 什么是 Linux? 为了回答这个问题, 需要首先谈论一下自由软件基金会以及开放源代码软件的理念。

假设您非常喜欢本书,希望所有的朋友都拥有一份。您会怎么做呢?您可以购买一整套书,然后送给您的朋友(这并不是一个坏想法,特别是如果您希望给您的朋友留下深刻印象的话)。当然,这样做需要花费大量的钱。但是,每个人都会收到一本真实的印刷书籍,而且至少您会觉得您的钱换来了一些东西。

另外还有一种方法,您可以复印本书。例如,您可以影印本书 30 本并将它们送给您的朋友。这样将给您节省一些钱,但是,相对于原版,影印本就没有那么完美。另外,进行影印、整理、装订并分发这些影印本也要花费大量的时间和精力,而且当您的朋友收到它们时,他们知道自己收到一个次等的产品。更糟糕的是,如果您的朋友希望自己复印本书,那么质量将会更次,因为影印的影印远没有原版那么好。

现在,假定您正在阅读一本电子版本的,而且希望和朋友共享这本书。您所需做的全部工作就是复制一些文件并将它们通过电子邮件发给您的朋友,或者刻录一张 CD 送给您的朋友。这样做不但便宜(可能还免费),而且副本和原版完全相同。另外,您的朋友还可以方便地对副本进行复制。

在进行此类复制时要确保该操作是合法的,但是这里我们先不考虑合法问题。从道德上讲,复制和分发电子数据(书、软件、音乐、视频等)是正确的还是错误的呢?

这个问题并不容易回答。它完全取决于您的观点,而每个人都拥有自己的观点。我可以告诉您一件事情:因为电子副本非常便宜和可靠,所以我们总是倾向于贬低电子格式内容的价值。例如,考虑一下购买本书需要花费多少钱,花费同样的钱购买一张包含本书的 CD,您是否愿意呢?或者花费同样的钱通过网络在线阅读本书呢?

因为复制(或者偷窃)电子数据非常容易,所以人们认为电子数据没有那么贵。基于该原因,软件公司在发行程序时,在没有严格许可证协议的情况下一般都会非常谨慎。许可证协议可以限制程序的复制和修改。

但是,您是否相信有一些天才的程序员,他们以这样一种方式发行软件,即鼓励复制软件?这种方式会让世界向好的一面改变吗?在 20 世纪 80 年代早期,一个名叫 Richard Stallman 的爱梦想的人也在思考这一问题。

Stallman 自 1971 年一直在麻省理工学院人工智能(Artificial Intelligence, AI)实验室工作。人工智能实验室在与他人共享软件方面拥有悠久的历史,不仅在实验室内部,而且还与其他组织共享软件。但是,在 1981 年,情况发生了变化,许多人离开了人工智能实验室,加入到一个新成立的公司。主计算机变了,操作系统也换成了一个专有的系统。Stallman 发现他们工作的环境变了,他和他的同事已经没有查看及修改操作系统的权利了。

碰巧 Stallman 不仅是一名程序员专家,还是一名有思想的社会批评家,他认为将操作系统换成一个专有操作系统是对他作为一名创造者的社会权利的限制。以他的话说,一个“专有软件社会系统,在这里面不允许对软件进行共享或者修改”不仅“反社会”,而且还是“不道德”并且“错误”的。他认为,这样的系统会在程序员和软件公司之间创建不健康的权利斗争。

而这种问题的解决方法就是从头开始创建大量编写出色、有用的软件,并且这些软件可以自由发行。

1984 年 1 月,Stallman 辞职,投身于该项目。在短时间内,他吸引了一小批程序员,并且在 1985 年,他们启动了一个称为自由软件基金会(Free Software Foundation, FSF)的组

织。Stallman 的指导原则就是“计算机用户应该能够自由地修改软件以适应自己的需求，并且自由共享软件，因为帮助他人是基本的社会责任。”Stallman 相信程序员的天性是希望共享他们的工作，而且当他们这样做时，所有人都会受益。

重要的是要理解 Stallman 所谈论的“自由软件”的含义，他所指的不是花费，而是自由。任何人都可以对自由软件进行检查、修改、共享以及发行。但是，根据 Stallman 的意思，如果有些人对他们的服务收费，或者要其他人对软件发行付费，这也没什么不对。Stallman 这样解释自由软件，即“谈话免费，但是啤酒不免费。”

因为单词“自由”既指自由又指花费，所以多少会有点模糊。因此，为了避免任何潜在的混淆，自由软件现在称为开放源代码软件(Open Source Software)。

这一名称取自一个编程术语。当程序员创建程序时，他所编写的实际指令称为源代码(Source Code)，简称为源(Source)或者代码(Code)。

(例如，假设您在和英国女王谈话，她抱怨她所使用的程序在播放 CD 音乐时一直有爆破声。她说：“如果我有源代码，那么我就可以叫 Charles 来修复这个错误。”“不用担心，”您回答到，“我知道在那里获取该源代码，我将把 URL 传给您。”)

因为任何计算机的核心都是操作系统，所以 Stallman 为 FSF 制定的第一个目标就是创建一个可以自由共享并且可以被任何人修改的操作系统。为了使新操作系统能够自由发行，Stallman 意识到操作系统必须从头开始编写。

Stallman 希望 FSF 的产品能够平稳地融入到目前流行的编程文化中，因此他决定新的操作系统应该与 Unix 兼容，也就是说它看上去就像 Unix 系统一样，并且还能够运行 Unix 程序。按照他曾经工作过的编程社区的传统，Stallman 为他还没有构建的操作系统选择了一种古怪的名称。他将这一操作系统称为 GNU。

名称含义

GNU

GNU 是 Stallman 选择用来描述自由软件基金会所开发的一个完全类 Unix 操作系统的项目的名称。名称本身是“GNU's Not Unix”只取首字母的缩写词，发音为“ga-new”(它与打喷嚏时的声音很押韵)。

注意，在表达式“GNU's Not Unix”中，单词 GNU 可以无限地扩展下去：

```
GNU
(GNU's Not Unix)
((GNU's Not Unix) Not Unix)
(((GNU's Not Unix) Not Unix) Not Unix)
((((GNU's Not Unix) Not Unix) Not Unix) Not Unix)
```

因此，GNU 实际上是一个递归的缩写词(递归指根据其自身来进行定义的内容)。

2.7 GNU 宣言摘录

正如上一节中所提到的，Richard Stallman 作为自由软件基金会的创始人，不仅是一名

程序员,还是一名有教养的社会批评家,而且他对未来的见解对全世界拥有极大的影响力。

在FSF成立不久,Stallman撰写了一篇小论文。在这篇论文中,他解释了促进自由软件思想的原因。他称这篇论文为GNU宣言。

他的基本思想——所有的软件应该自由共享——听起来非常天真,但是,随着Internet的兴起,开放源代码软件(即Stallman所谓的“自由软件”)的开发和发行已经成为我们这个世界一个重要的经济和社会力量。差不多有数万个程序可自由使用,它们对世界的贡献非常巨大,难以用金钱衡量,而且编写这些程序的程序员也对此非常兴奋。

在下一节中我们将继续讨论这一理念。在这之前,我想先从这篇1985年的原始论文中摘录几段。

作为一名哲学家,Stallman并不是一名重量级的人物。他的公开言论没有像其他有名的宣言,例如95 Theses(马丁·路德,1517)、Manifesto of the Communist Party(卡尔·马克思和弗雷德里希·恩格斯,1848)或者The Playboy Philosophy(Hugh Hefner,1962-1966)那么深奥微妙。然而,自由软件基金会的工作非常重要,而且在今天来说,它对我们的文化仍然具有重要的贡献。基于这一原因,大家可能会对Stallman在1985年所发表论文的一些片段感兴趣。

(题外话:当Stallman在麻省理工学院工作时,他开发了Emacs文本编辑器。如果您使用过GNU版本的Emacs(Linux中就使用了GNU版本的Emacs),那么您可以通过启动Emacs,然后输入命令<Ctrl-H> <Ctrl-P>显示完整的GNU宣言。)

GNU 宣言摘录

“我认为:如果我喜欢一个程序的话,那我就应该将它分享给其他喜欢这个程序的人。这句话是我的座右铭。软件商想各个击破用户,使他们同意不把软件与他人分享。我拒绝以这种方式破坏用户之间的团结。我的良心使我不会签下一个不开放的合约或是软件许可证协议。在麻省理工学院人工智能实验室工作的多年时间里,我一直反对这样的趋势与冷漠,但是最后事情糟糕到:我无法在一个处理事情的方法与我的意愿相违背的机构呆下去。”

“为了能继续使用电脑而不感到羞愧,我决定将一大堆自由软件集合在一起,从而使我可以不必再使用不自由的软件。因此,我辞去了人工智能实验室的工作,不给麻省理工学院任何法律上的借口来阻止我把GNU送给其他人……”

“很多程序员对系统软件的商业化感到不悦。这虽然可以使他们赚更多的钱,但是这使他们觉得自己与其他程序员处于对立状态,而不是同志之间的感觉。程序员对友谊的最基本表现就是共享程序,而当前的市场运作基本上禁止程序员将其他程序员作为朋友看待。软件购买者必须在友谊和守法之间做一选择。自然地,有很多人选择了友谊。但是那些相信法律的人常常没办法安心地做这一选择。他们会变得愤世嫉俗,认为写程序只不过是赚钱的一种方法而已。”

“复制全部或者部分程序对程序员来说就和呼吸一样是自然有益的事。复制软件就应该这么自由……”

“从长远来看,免费提供软件是迈向大同世界的一步,在那个时代中,没有人再为了生计而努力工作。在每周10小时的必要劳动(如立法、家政服务、机器人修理和行星观察)之后,人们自由地参与各种自己感兴趣的活动,例如编程。那时候就不必靠写程序来过活了……”

2.8 GPL 和开放源代码软件

多年以来, Stallman 和许多其他支持自由软件基金会的程序员们一起勤奋工作, 创建了大量的开放源代码软件。实际上, 正如前面所提及的, Stallman 是 Emacs 文本编辑器的原创者。

今天, FSF 已经不再是倡导开放源代码软件的唯一组织。实际上, 这样的组织已经有许多。但是, FSF 一直是领导者之一, 不仅开发了 Emacs, 而且还开发了一种 C 编译器(gcc)、一种 C++编译器(g++)、一个功能强大的调试器(gdb)、一个 Unix shell(Bash), 还有许多其他工具。所有这些软件——它们都是 GNU 项目的一部分——都被遍布全球的用户所使用, 而且被认为是拥有最高质量的工具。

在 20 世纪 80 年代后期, Stallman 在创建自由软件方面已经积累了一定的经验, 他认为如果要创建一个大型的自由软件, 那么他需要一个合适的许可证协议, 并在这个许可证协议之下发行该软件。为了这个目的, 他提出了非盈利版权(copyleft)这一理念(这一名称取自 Stallman 的一位朋友在 20 世纪 80 年代中期发给 Stallman 的一封信, 在信封上这位朋友写了几个诙谐的格言, 其中一个就是“Copyleft—all rights reversed”)。

在软件社区中, 传统的版权用来限制软件的使用。而非盈利版权的目的是相反的。以 Stallman 的话来说: “非盈利版权的中心思想就是授予任何人运行程序、复制程序、修改程序以及发行修改后程序的权限——但是不能在自己修改后的软件上添加限制。”

为了实现非盈利版权这一思想, Stallman 提出并编写了通用公共许可证(general public license, GPL), 并于 1989 年发布。GPL 本身相当复杂, 因此我不详细介绍它。基本而言, 当在软件上应用 GPL 协议时, 允许任何人发行该软件、查看其源代码、修改该软件并发行修改后的软件。此外, GPL 协议要求任何重新发行软件(包括修改后的版本)的人, 都不能剥夺软件的使用自由或者添加自己的限制——这是 GPL 的关键部分。

GPL 协议确保无论何时、何人使用自由软件创建一个新产品时, 新产品都必须在 GPL 协议下发行, 不能在其他协议下发行。实际上, 这意味着如果某人以自由软件为基础, 修改了自由软件后再重新发行它, 那么这个人也必须发布源代码。

GPL 协议变得非常流行, 多年以来又发展了许多相似的许可证协议。实际上, 非盈利版权许可证协议和 Internet(允许全球各地的程序员共享软件以及一起工作)的结合使共同创建(即所谓的开放源代码运动)的规模空前繁荣。

开放源代码运动是如此的重要, 因此它对编程以及全世界的影响怎样夸大也不过分。通过一个例证大家就可以看到它的重要性, 如果所有的开放源代码软件都突然消失, 那么我们所知道的 Internet 也会在一瞬间消失(全球大多数 Web 服务器都由一个称为 Apache 的开放源代码程序运行)。

在 Unix 世界中, 开放源代码运动的影响有深远的历史意义。这种说法的原因有许多, 最重要的一条是要不是 FSF 和 GPL, 以及它们对编程文化进行的变革, 您和我或许永远都不会听说 Linux。

2.9 20 世纪 70 年代的 Unix: 由贝尔实验室转向 Berkeley

在第 1 章的开头,我解释过第一版的 Unix 于 1969 年在新泽西州 AT&T 公司的贝尔实验室开发。在 20 世纪 70 年代早期,Unix 由一群来自贝尔实验室的人们重新进行了编写和增强。

1973 年,一个 Unix 开发支持小组成立。这年后期,Ken Thompson(Unix 的两个主要开发人员之一)在一个计算机专业会议上提交了有关 Unix 的第一篇论文。这激起了人们对 Unix 的兴趣,6 个月之后,运行 Unix 的站点数量由 16 个增加到 48 个。

1974 年 7 月,Thompson 和 Dennis Ritchie(另外一个 Unix 的主要开发人员)在 *Communications of the ACM*^{*}上发表了一篇论文“*The UNIX Time-Sharing System*”,*Communications of the ACM* 是全球订阅最广泛的计算机科学杂志。这是 Unix 第一次被公开详尽地介绍给全世界,而全世界也很快响应起来。大学里面的研究所以及公司里的研究人员请求获取 Unix 的副本,并在自己的计算机上运行 Unix。在一些场合中,他们移植 Unix(也就是说他们改编 Unix),以运行在一种新类型的硬件上。

除了贝尔实验室外,Unix 开发最重要的核心就是加利福尼亚大学伯克利分校的计算机科学技术系。1974 年,加利福尼亚大学伯克利分校的一名教授 Bob Fabry 获得了一份 AT&T 公司第 4 版的 UNIX 副本,伯克利分校的学生们开始对这一系统进行增强。1975 年,Ken Thompson 前往伯克利分校进行一年的交流访问,充当了伯克利分校 Unix 开发的催化剂。

在同一年,AT&T 公司正式开始将 UNIX 许可给大学(大家应该还记得,AT&T 公司一直坚持“UNIX”必须用大写字母拼写,当具体指 AT&T 公司的 UNIX 时,我采用了这种方式)。

在这一时期,伯克利分校的一名研究生 Bill Joy 开始对 Unix 感兴趣。尽管在那个时候还没有人知道 Joy,但是 Joy 的工作对 Unix 产生了深远的影响。

Joy 所做的工作为伯克利分校今后成为 Unix 的主要参与者打下了坚实的基础。伯克利分校创建并发行了自己版本的 Unix。实际上,目前几个重要的 Unix 都是伯克利分校的 Unix 以及 Bill Joy 工作的直接分支。

另外,Joy 还是一名技能丰富并且多产的程序员,在几年的研究生课程学习期间,他独自开发了大量重要的软件。即使在今天,现有的 Unix 系统也没有一个不受 Joy 在 1975 年至 1982 年期间在伯克利分校所做工作的重要影响。1982 年,Joy 成为 Sun 公司的共同创始人。在这个时代,Sun 公司是全球最重要的 Unix 公司之一。

或许我可以以这种方式描述 Joy 工作的重要性:虽说 Joy 在伯克利分校工作已经过去

^{*}1 个月之后,在 1974 年 8 月,我在这个期刊上发表了第一篇技术论文。那时,我还是加拿大 Waterloo 大学的一名大学生,而且我赢得了 ACM 举办的 George E. Forsythe Student Paper Competition 竞赛。这篇论文的题目是“*A New Technique for Compression and Storage of Data*”。

为了防止您感到迷惑,介绍一下 ACM。ACM 是计算机科学家的主要专业协会。这一名称可以追溯到 1947 年,代表 Association for Computing Machinery(计算机协会)。

这里有一些有趣的事情。如果您查看 CACM(该期刊通常这样称呼的)的过期期刊(back copy),您就会发现含有第一篇 Unix 论文的 1974 年 7 月的期刊经常丢失,毫无疑问,这是那些过分狂热的纪念品收藏者的杰作。但是,1974 年 8 月的期刊(发表我的第一篇论文的那一期期刊)通常不会丢失。

了 20 多年,然而,本书中有几章讨论的程序都是 Joy 在那个时代开发的。这些程序包括 vi 编辑器(第 22 章),编写于 1976 年;以及 C-Shell(第 11-14 章),编写于 1978 年。

(这是一个令人吃惊的事实,今天仍在使用的两个最流行的 Unix 文本编辑器,都是在很久以前编写的,vi 由 Bill Joy 于 1976 年编写;Emacs 由 Richard Stallman 于 1975 年编写。)

1977 年, Bill Joy 装配了第一版的伯克利 Unix,而我也是在这一年开始使用 Unix 的(那时,我是圣地亚哥加利福尼亚大学计算机科学专业的一名研究生)。

Bill Joy 装配的系统称为伯克利软件套件(Berkley Software Distribution),后来简写为 BSD。后来, Joy 在伯克利分校外总共装配了 30 多份副本。尽管这个数字看上去很小,但这一个成功,在 1978 年中期, Joy 装配了下一版本的 Unix——2BSD。

1979 年, AT&T 公司最终认识到了 UNIX 的潜能,宣布他们准备开始将 UNIX 作为一个商业产品进行销售。第一个商业版本的 UNIX 称为 UNIX System III(“System Three”)。很快,它就被 UNIX System V(“System Five”)取代。

到 1979 年,所有的 BSD 用户都被要求购买一个 AT&T 公司的许可证,而且每年 AT&T 公司都会提升许可证的价格。渐渐地, BSD 程序员开始难以忍受 AT&T 公司的束缚。

2.10 20 世纪 80 年代的 Unix: BSD 和 System V

第一版的 Linux 开发于 1991 年,我们稍后再介绍它。但是,为了理解 Linux 是如何登上历史舞台的,我们需要看看 Unix 在 20 世纪 80 年代发生了什么情况。具体而言,我希望解释 Unix 是如何演变进化的。到 20 世纪 80 年代末, Unix 有两个主要的分支: BSD 和 System V。

截止到 1980 年,美国东海岸 Unix(AT&T 公司的 UNIX)和西海岸 Unix(BSD)平分秋色,且都发展很快。

伯克利分校的程序员和 BSD 用户抱怨只是安装 BSD 还要向 AT&T 公司支付金钱。另一方面, AT&T 公司决定使 UNIX 成为一个成功的商业产品,只面向那些能够为许可证支付大量金钱的公司。

1980 年, Bob Fabry(我在前面提到的伯克利分校的教授)收到来自 DARPA(the U.S. Defense Advanced Research Project Agency, 美国国防部高级研究计划局)的一个大合同,开始开发 Unix。DARPA 已经建立了一个遍布全国的计算机网络,将他们主要的研究中心连接起来。这样一来,他们希望得到一个可以运行在不同类型硬件上的操作系统。伯克利分校接受的合约就是为这一目的开发 Unix 系统。

(题外话: DARPA 还是在 1965 年至 1988 年期间资助建立 Internet 的机构。在 1972 年之前,该机构的名称是 ARPA。因此, Internet 的祖先称为 Arpanet——一个大家可能已经听说过的名字。Arpanet 作为一个单独的实体,于 1989 年关闭。)

Fabry 一得到 DARPA 的合约,就建立了一个称为计算机系统研究小组(Computer Systems Research Group)或者 CSRG 的组织。那个时候 Fabry 并不知道, CSRG 会一直延续到 1994 年,且在这段时间中, CSRG 对 BSD 和 Unix 在全球的发展产生了重要的影响。

在 1980 年, Fabry 所关心的全部工作就是开发 BSD,而且它也是 CSRG 集中攻关的任务。几年之后,他们发行了许多版本的 Unix,所有版本的 Unix 都受到学术界和研究社区

的高度关注。BSD 用户的数量开始增长。到 1982 年, 4.1BSD 已经支持 TCP/IP, 从而使 BSD 系统开始成为 Internet 的基础。1983 年, 4.2BSD 发布, 这一版本的 Unix 非常流行, 在全球大约有 1000 份安装案例。

在商业领域, AT&T 公司朝一个完全不同的方向发展。AT&T 公司的目标就是将 UNIX 发展成为一个商业产品。1982 年, 他们发行了 UNIX System III, 这是第一个在贝尔实验室之外公开发行的 UNIX。1983 年, 他们发行了 System V, 这是第一个提供官方支持的版本。在同一时间, AT&T 公司将 3 个内部小组组合成一个部门, 创建了 UNIX 系统开发实验室。

到这一年末, System V 安装了 45 000 份。1984 年, 当 System V Release 2(SVR2)发行时, 大约安装了 100 000 份。

因此, 到 1985 年, Unix 流派主要有两个。可以肯定的是还有其他形式的 Unix, 但是这些 Unix 或者派生自 BSD, 或者派生自 System V*。

Unix 世界在 20 世纪 80 年代末主要有两大特征: Unix 的总体快速增长和不同类型 Unix 的增殖扩散。

图 2-1 列出了 20 世纪 80 年代及其后所使用的最重要的商业 Unix(少数几个操作系统今天仍在出售)。每种类型的 Unix, 毫无例外地都基于 BSD 或 System V, 或者基于二者。

名 称	公 司	BSD 还是 SYSTEM V?
AIX	IBM 公司	BSD + System V
AOS	IBM 公司	BSD
A/UX	苹果公司	BSD + System V
BSD/OS	Berkeley Software Design	BSD
Coherent	Mark Williams Company	System V
Dynix	Sequent	BSD
HP-UX	惠普公司	System V
Irix	Silicon Graphics	BSD + System V
MachTen	Tenon Intersystems	BSD
Nextstep	Next Software	BSD
OSF/1	DEC 公司	System V
SCO Unix	Santa Cruz Operation(SCO)	System V
Solaris	Sun 公司	BSD + System V
SunOS	Sun 公司	BSD
Ultrix	DEC 公司	BSD + System V
Unicos	Cray Research	System V
UNIX	AT&T 公司	System V
Unixware	Novell 公司	System V
Xenix	微软/SCO/Altos/Tandy	System V

图 2-1 最重要的商业 Unix 类型

* 实际上, 这一状况一直持续到 20 世纪 90 年代, 直到 Linux 的影响力日渐强大, 以及开放源代码运动对 Unix 世界产生了永久性的改变为止。

例如, 在本书的前两版(1993 年和 1996 年)中, Unix 有两种基本的变体: BSD 和 System V。

尽管所有重要的 Unix 都派生于 BSD 或者 System V, 但是它们都是不同的版本, 若干年以来, 混战和竞争一直持续不断。

在 20 世纪 80 年代后半期, 作为一名作家兼咨询顾问, 我记得这些阴谋和混乱。Unix 世界的主要特征就是不同公司之间结成联盟、破坏联盟、建立协会、解散协会以及一个接一个地提出技术建议, 所有这些都是为了“标准化” Unix, 从而主宰市场。

同一阶段, 在大学和研究机构中, Unix 用户无限制地递增。许多用户使用 BSD 的变体, 他们对 AT&T 公司对他们正在做的事情拥有局部控制不满。而解决方法就是从头开始重新编写 BSD 中基于 AT&T 公司 UNIX 的那一部分代码。

这一任务花费了很长一段时间, 但是伯克利分校的程序员都非常勤奋地工作, 每次都替换 BSD 中的一部分 UNIX 组件。创建一个完全独立的 BSD 版本是一个崇高的目标, 但是直到 1992 年这一任务才完成。

2.11 1991 年的 Unix: 等待中……

到 1991 年时, PC 已经面世 10 年。然而, 还没有哪一个 PC 操作系统能够吸引黑客——黑客是那种为了兴趣而剖析软件并修改软件的程序员。DOS 操作系统(适用于 PC)相对简单, 不能令人满意。苹果公司的 Macintosh 操作系统较好, 但是机器本身对于计算机业余爱好者来说过于昂贵。

Unix 就成为了最佳的选择。毕竟, 世界上一些最好的黑客多年以来一直在使用 Unix。但是, BSD(到那时为止)还不能在 PC 上运行, 而商业版本 Unix 的价格又令人望而却步。

计算机科学教授也面临着一个相似的问题。AT&T 公司的 UNIX 对讲授操作系统课程来说是一个不错的工具, 只是在 1979 年, AT&T 公司改变了他们的政策。在这一年, 即从 AT&T 公司的 UNIX Seventh Edition 开始, AT&T 公司不允许其公司之外的人查看 UNIX 的源代码。

AT&T 公司的新政策意味着已经不能再使用 UNIX 作为教学工具了, 因为操作系统课程的学生需要查看源代码, 从而明白系统是如何运转的。

计算机科学教授所需的操作系统是一个免费、直接可用的操作系统, 一个适合于教学而且可以查看源代码的操作系统。

Andrew Tanenbaum 就是这样一名教授, 他在阿姆斯特丹的 Vrije Universiteit 大学任教。他购买了一台 IBM 公司的 PC, 并且准备从头开始构建自己的操作系统。新操作系统与 Unix 极其相似, 但是规模非常小, 因此 Tanenbaum 称它为 Minix(“minimal Unix”)。

Minix 不包含所谓的 AT&T 公司的源代码, 这意味着 Tanenbaum 可以根据自己的意愿发行新的操作系统。第一版本的 Minix 于 1987 年发行, 与 UNIX Seventh Edition 兼容。它主要用于教学目的, 作为教学工具, 它是一种很好的操作系统(但并不十分完美)。

多年以来, Minix 是程序员学习操作系统以及体验操作系统的最佳工具。全世界都开始使用它, 特别是大学里的人们。同时, 一大批狂热的志愿者开始根据自己的需要和意愿修改 Minix。

然而, Minix 无法满足人们的期望。大量的程序员喜欢它, 并且希望对官方版本进行

增强,但是 Minix 版权的拥有者 Tanenbaum 否决了大部分请求。他坚持 Minix 应该是一个简单的操作系统,适合于教学需要。

Minix 用户不高兴了。当 Richard Stallman 于 1985 年成立自由软件基金会时,他激发了全世界程序员的热情。他们渴望一个健壮、自由的操作系统,在这个操作系统上他们可以释放他们的集体能量。

现在,Stallman 已经计划以一个自由且与 Unix 兼容的操作系统作为他的第一个主要项目。他甚至还为它起了一个名字 GNU。到 1991 年,Stallman 和 FSF 已经创建了大量高质量的自由软件,但是离 GNU 本身还相差甚远。

正如前面所述,Unix 由两种不同类型的程序构成,即内核和其他程序,后一部分程序又称为“实用工具”。到 20 世纪 80 年代末,FSF 已经编写了许多重要的实用工具,包括一个 shell。但是,他们还没有一个成型的内核(操作系统的核心)。

Stallman 和 FSF 程序员工作了一段时间,但是 GNU 内核——Stallman 称之为 HURD,还远远没有准备好。实际上,HURD 的工作从 1990 年才真正开始,多年以来,全世界的程序员每天晚上都熬到很晚,悲痛地唱着:“总有一天,我的 HURD 就会实现……”

总而言之,到 1991 年,PC 已经面世 10 年,数千名程序员、学生和计算机科学家都强烈地期望拥有一个开放源代码的类 Unix 操作系统。

但是,AT&T 公司商业化了 UNIX。BSD 还可用,但是 BSD 受 AT&T 公司许可证协议的制约。Minix 可以用,但是没有那么完美。尽管源代码可以使用,但是它要求许可证,而且还无法自由共享。另一方面,GNU 在 GPL 的非盈利版权下发行,这一点非常伟大,但是,内核——操作系统最重要的部分——还需要很长的时间才能完成。

就像命中注定一样,一名年轻的芬兰学生 Linus Torvalds 刚刚启动了一个项目:只是为了乐趣,他决定编写自己的操作系统内核。

然而,Torvalds 没有想到,不到十年时间,他的个人操作系统成为历史上最重要的开放源代码事业、一个革新的项目,而且在时机成熟时,吸引了数十万程序员参与其中,简直可以说改变了世界。

名称含义

Hurd

在讨论内核时,我解释过大多数操作系统或者使用一个单内核(一个单独的大型程序),或者使用一个结合了大量小型程序(称为服务器)的微内核。对于 GNU 来说,Richard Stallman 选择使用一个称为 Mach 的微内核,并结合了一组服务器,他称这些服务器为“Hurd”(这一名称由 GNU 的主要内核程序员 Thomas Bushnell 杜撰而来)。

严格地说,GNU 内核应该描述为运行在 Mach 之上的 Hurd(服务器)。但是,在普通应用中,通常将整个内核称为 Hurd。

大家已经知道,当 Stallman 为自由软件基金会的 Unix 选择名称时,他选择了 GNU:一个“GNU's not Unix”的递归首字母缩写词。当要为内核命名时,Stallman 的意见又占了上风。

名称 Hurd 代表“HIRD of Unix-Replacing Daemons”(在 Unix 中,“daemon”指一种在后台独自运行的程序)。名称 HIRD 代表“HURD of Interfaces Representing Depth”。

因此,Hurd 是一个间接的递归首字母缩写词(说实话,这可能是您一生中唯一遇到的一个)。

2.12 真命天子: Linus Torvalds

Linus Torvalds 是一个在合适的时间、合适的地点提出合适的思想的最典型的人。

1991 年, Linus(发音为 “Lee’-nus”)是 Helsinki 大学计算机科学系的二年级学生。与其他数以万计的喜欢摆弄编程的学生一样, Linus 阅读了 Andrew Tanenbaum 的书 *Operating Systems: Design and Implementation*, 该书解释了 Minix 的设计原则。作为该书的一个附录, Tanenbaum 在书中包含了其操作系统的 12 000 行源代码, Linus 花费了许多时间来研究这些代码(此时, 您可以思考一下阅读 12 000 行代码意味着什么, 并且与您在本书附录中看到的信息类型进行对比)。

像其他许多程序员一样, Linus 希望一个自由(开放源代码)版本的 Unix。但是 Linus 又不像其他程序员, 他不愿意再等待了。

1991 年 8 月 25 日, Linus 向 Usenet 讨论组(也是 Minix 的论坛, comp.os.minix)发了一个消息。回顾过去, 这个短消息已成为 Unix 世界的历史文档之一, 因此我将它整个展示出来(见下页)。当您阅读它时, 要知道英语是 Linus 的第二语言, 所以这篇文章写得并不正规。

很清楚, Linus 只是试图通过构建自己的操作系统来寻找乐趣。他意识到自己的主要工作是编写内核, 因为在很大程度上, 自由软件基金会已经提供了各种实用工具, 在编好内核之后, 他可以直接使用这些实用工具。

```

LINUX TORVALDS ANNOUNCING HIS NEW PROJECT...

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki

Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby,
won't be big and professional like gnu) for
386(486) AT clones. This has been brewing since
april, and is starting to get ready. I'd like any
feedback on things people like/dislike in minix,
as my OS resembles it somewhat (same physical
layout of the file-system (due to practical
reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40),
and things seem to work. This implies that I'll
get something practical within a few months, and
I'd like to know what features most people would
want. Any suggestions are welcome, but I won't
promise I'll implement them :-).

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has
a multi-threaded fs [file system]. It is NOT
portable (uses 386 task switching etc), and it
probably never will support anything other than
AT-harddisks, as that's all I have :-).

```

1991 年 9 月, Linus 发行了第一版的内核, Linus 将这个内核称为 Linux。Linux 通过 Internet 发行, 而且在极短的时间内, Linus 开始一个接一个地发行新版本的 Linux。全世

界的程序员们开始加入到 Linus 的行列, 首先数十人, 然后数百人, 最终达到了数万人。

这里有一件有趣的事情需要提及, 那就是 Linus 选择使用单内核来设计 Linux。由 Andrew Tanenbaum 设计的 Minix 使用的是微内核。在 Linux 开始引起注意后不久, Tanenbaum(一位著名的教授)公开批判 Linus(一个没有名气的学生)的这种设计决策。即使在今天, 在 Linux 内核成为历史上最成功的 Unix 时, Tanenbaum 仍然批评 Linus 使用了单内核(就个人而言, 我阅读过 Tanenbaum 的观点, 我认为他是错误的)。

最终, Linux 变得非常流行, 运行在每一种类型的计算机上: 不仅运行在 PC 上, 而且还运行在安装有小型内置处理器的手持设备以及大规模并行超级计算机集群(世界上最快、功能最强大的系统)上。实际上, 到 2005 年, 几乎世界上每个计算领域都存在着运行某种类型的 Linux 的机器, Linux 成为世界上最流行的操作系统(Windows 的应用更加广泛, 但是 Linux 更加流行)。

为什么 Linux 如此成功呢? 原因有 4 个方面, 而且它们都与 Linus 本人相关。

首先, Linus Torvalds 是一名技能和知识极端丰富的程序员。他是一名高效的工人, 而且他热爱程序。换句话说, 他正好是那种准备自己编写操作系统内核的人。

其次, Linus 是一名无止境的完美主义追求者。Linus 致力于他的工作, 当遇到问题时, 他会放弃休息, 直至找到合适的解决方案。在 Linux 的早期, Linus 快速地对内核进行修改, 有时候甚至每天发行一个新内核, 这很不寻常。

第三, Linus 拥有令人喜爱的个性。人们描述他是一个低调、谦逊的家伙, 一个诚实的高尚人物(参见图 2-2)。无论是当面还是在线, Linus 与他人相处得都非常融洽。正如他在一次采访中的评述: “与 Richard Stallman 不同, 我真的没有什么要说的。”



图 2-2 Linus Torvalds

1991 年, Linus 成立了一个新项目来创建一个新的操作系统内核, 现在我们称这个内核为 Linux 内核。Linux 内核是各种 Linux 操作系统的基础, 从而使 Linus 从事的项目成为了历史上最重要的一个软件项目。大家可以从上述照片看出, Linus 并不十分在意个人形象。

第四, 也是最重要的一点, 就是 Linus 拥有使用 Internet 的天分, 可以将编程天才的智慧通过 Internet 融合在一起: 成千上万人志愿参与修改及扩展 Linux 内核。

Linux 的规模有多大呢? 今天, Linux 内核总共包含 17 000 多个文件, 代码有数百万

行。每天，志愿程序员都提交数百个补丁(修改或者改正)。与此相比，最初的 0.01 版只包含 83 个文件，代码总共还不到 9000 行。

如此众多的人一同参与这项任务也给 Linux 创造了难得的机遇，有助于他处理新发现的问题。例如，当新硬件出现时，总会有一些具有相应知识的人为新设备编写能在 Linux 下运行的代码。

最初，Linux 做了一个策略决定，即以 GNU GPL 的名义发布 Linux 内核。事实最终证明这一决定非常重要，因为它鼓励程序员自愿参与到其中。他们知道他们所做的任何事情都将与全世界的其他程序员自由共享。另外，GPL 还确保任何最终使用 Linux 内核的操作系统都将以 GPL 的名义发布，因此这些操作系统也都自由发行。

从一开始，Linux 就尽可能快地发布新版本的内核。通常，程序员都喜欢持有新版本的软件，这样他们可以彻底地测试它，并且在程序向大众发布之前尽可能地修复各种 bug。

Linux 的天才在于他意识到，由于他将软件发行给大量的爱好者，因此可以有许多的测试以及阅读新代码，致使 bug 很快就被发现。这一思想收录在所谓的 Linux 法则中：“Given enough eyeballs, all bugs are shallow(让足够多的人阅读源代码，错误将无所遁形)。”

Linux 发行新版本内核的速度比以往任何人都要快，bug 也会很快被确认和修复——通常是在数小时内。这样的结果使 Linux 飞快地发展和完善，比历史上任何一个主要软件项目都要快。

名称含义

Linux

Linux 的正确发音方式要有 “Bin'-ex” 押韵。

从一开始，Linus Torvalds 就非正式地使用名称 Linux，“Linux” 是 “Linus' Minix” 的缩写。但是，当 Linus 发布第一个公开版本的内核时，他实际上计划使用名称 Freax(“free Unix”)。

当 Linus 发行第一版的内核时，另一名程序员 Ari Lemmke，说服 Linus 将文件上传到一个由 Lemmke 运营的服务器上，从而使编程爱好者可以通过一个称为 “anonymous FTP” 的系统方便地访问文件。Lemmke 对名称 Freax 不满意，因此，当他在创建存放这些文件的目录时，他将这个目录命名为 Linux，Linux 名称由此而来。

2.13 Linux 发行版

严格地讲，Linus Torvalds 和 Linux 项目所创建的只是一个内核，并不是一个完整的操作系统(而且现在情况依然如此)。当 Linux 内核最初发布时，为了使用它，您需要是一名 Unix 专家，因为您必须为它寻找所需的各种组件，并将它们组装在一起才能形成一个操作系统。

但是，在 Linux 第一版内核发布 5 个月之后，其他人就开始提供基于 Linux 内核的操作系统。我们称这种操作系统为 Linux 发行版(distribution，有时候简写为 distro)。

可以想象，刚开始的少数几个 Linux 发行版深受 Linux 社区的欢迎，但遗憾的是，它

们没有得到很好的维护。但是, 1993年7月, 一个名叫 Patrick Volkerding 的程序员宣布了一个新的发行版, 并称之为 Slackware。这一名称来源于单词 “slack”, 指通过实现个人目标所获得的愉快和满足的感受(更多信息请查阅 Church of the SubGenius)。

Slackware 非常成功, 因为多年以来 Volkerding 一直设法维护它。现在, 它是最古老的 Linux 发行版, 且仍然在积极地发展。实际上, Slackware 如此流行, 以至于发展成为一个完整的发行版家族, 并拥有自己的支持小组。

现在, 现代的 Linux 发行版都提供一个完整的产品: 内核、实用工具、编程工具以及至少一个 GUI。在我编写本书时, Linux 发行版差不多已有数百个(如果您愿意, 一旦您精通了 Unix, 就可以自己制作一个 Linux 发行版)。

在结束本节内容之前, 我希望确保您已经理解了单词 “Linux” 拥有两层含义: 首先, “Linux” 指一个内核, 无数在 Linux 项目中工作的程序员的一个产品。

其次, “Linux” 指任何基于 Linux 内核的操作系统名称。这是大多数人谈话时使用这个单词的方式, 而且也是本书使用这个单词的方式。

大多数 Linux 发行版都使用自由软件基金会的 GNU 实用工具。基于这一原因, FSF 坚持基于 Linux 的操作系统实际上应该称为 GNU/Linux。我不知道有谁这样做, 但是您应该记住, 以防您在 Unix 吧闲逛时碰巧遇上了 Richard Stallman。

2.14 BSD 发行版

正如前面所述, BSD 是 20 世纪 80 年代两大 Unix 主流之一(另一个是 System V)。BSD 由加利福尼亚大学伯克利分校计算机科学系开发, 自 1980 年起, 开发由计算机科学研究小组(Computer Science Research Group, CSRG)管理。

20 世纪 80 年代末, BSD 狂热分子开始为 AT&T 公司对 Unix 的商业化运作感到不满。因为 BSD 包含有 AT&T 公司 UNIX 的特定专有部件, 而且 CSRG 随操作系统一起发布源代码, 所以每个 BSD 用户都被迫从 AT&T 公司购买一份昂贵的源代码许可证。

基于这一原因, CSRG 制定了一个目标: 完全重写所有基于 AT&T 公司的 BSD 部件。这样做可以使 BSD 永远免于受 AT&T 公司律师(以及 AT&T 公司市场部门)的制约。

1989 年, CSRG 提供了第一个全部开放源代码的 BSD 发行版, 并将它称为 Networking Release 1(更正式的称呼是 4.3BSD NET/1)。但是, NET/1 包含的大部分网络工具并不与 AT&T 公司的 UNIX 独立, 因此仍然不是完全的 BSD 操作系统。

为了创建一个真正独立的 BSD 版本, 必须重新编写数百个 AT&T 公司的实用工具和编程工具。通过使用 Internet, CSRG 恳求外部的程序员帮忙, 不到一年半时间, 所有的 AT&T 公司实用工具就由全新的程序替换了。

经过对代码的仔细检查, 结果显示除了 6 个内核文件外, BSD 中的所有内容都完全属于自己了。重新编写这 6 个文件是一项大工作, 因此在 1991 年, CSRG 先发行了一个新版本的 BSD, 它几乎与 UNIX 无关。他们称它为 NET/2。

1992 年, 一个名叫 Bill Jolitz 的程序员重新编写了最后 6 个有问题的文件, 并使用它们为 PC 创建了一个新版本的 BSD。他称这个操作系统为 386/BSD, 并且开始通过 Internet 发行。

这是一个巨大的成就。毕竟这些年来，伯克利分校的神圣目标——一个与 AT&T 公司 UNIX 无关的操作系统——实现了。最终，BSD 可以在全球作为开放源代码软件自由发行。

这一点非常重要，因为 BSD 包含了一些历史上最佳的 Unix 软件。就今天来说，世界上没有几个 Unix 不包含 BSD 代码。实际上，大多数 Linux 发行版也使用大量的 BSD 实用工具。基于这一原因，386/BSD 的创建是 Unix 发展历史的一个里程碑。

在很短的时间内，386/BSD 就流行起来，而且维护它的人数日益递增。但是，Jolitz 拥有一份全职工作，他没有时间处理所有提交的补丁和增强。基于这一原因，一组志愿者接管了这项工作。该小组所做的第一件事就是将该系统重新命名为 FreeBSD。

刚开始，FreeBSD 只能运行在 PC 硬件上，这适合大多数用户的需求。但是，也有一些用户希望在其他类型的机器上运行 BSD，因此人们成立了一个新的小组，该小组的目标就是尽可能地将 FreeBSD 移植到许多其他类型的计算机上。这个新小组提供的版本称为 NetBSD。

20 世纪 90 年代中期，NetBSD 小组又分出另一个小组，该小组主要关注安全和密码学问题。这个小组的操作系统称为 OpenBSD。

可以看出，BSD 世界只有 3 个主要的发行版(FreeBSD、NetBSD 和 OpenBSD)，这与 Linux 世界差别很大。在 Linux 世界里，差不多拥有数百个不同的发行版。

到现在为止，您或许会觉得奇怪，FreeBSD 是完全的开放源代码软件，也可以通过 Internet 自由发行，那么 Linux 是如何成为快速流行起来的 Unix 版本呢？

原因有两方面。首先，Linux 基于 GNU GPL 名义发行，而 GNU GPL 协议鼓励共享。因为 GPL 禁止任何人使用 Linux 创建及发行专有系统，所以对 Linux 所做的任何事情都属于全世界。

BSD 许可证远没有 GPL 严格。在 BSD 许可证之下，允许使用部分 BSD 创建新产品而不共享该新产品。当这种情况发生时，全世界很大程度上无法从新产品上获取好处，也不能使用和修改新产品。基于这一原因，许多程序员喜欢使用 Linux。

(另一方面，因为 BSD 许可证非常灵活，所以 FreeBSD 广泛应用在大量不同类型的机器和设备上，以及全球众多的 Web 服务器上。)

Linux 比 FreeBSD 更成功的第二个原因在于 Linux 的发行时机上。回顾过去，非常明显，在 20 世纪 80 年代末时，全世界的程序员都强烈渴望一个完全开放源代码的 Unix。因为 Linus Torvalds 在 1991 年发行了 Linux 内核，而 386/BSD 直到 1992 年才发行，所以 Linux 在发行时间上占得先机。结果是，Linux 能够吸引大量的程序员，而这些程序员正在等待全球第一个自由版本的 Unix。

现在您应该知道为什么本书称为 *Harley Hahn's Guide to Unix and Linux*，而不是 *Harley Hahn's Guide to Unix and FreeBSD* 了。

(而且现在您也应该知道我为什么将 Linus Torvalds 描述为在合适的时间、合适的地点提出合适思想的家伙了。)

2.15 您应该使用什么类型的 Unix

面对所有不同类型的 Unix，您应该使用什么类型的 Unix 呢？

答案(有数个)实际上相当简单。但是,在回答之前,我先告诉您一点,即实际上基本的 Unix 就是基本的 Unix: 如果您知道如何使用一种类型的 Unix, 那么您就知道如何使用所有的 Unix。特别是在阅读本书时, 您使用何种类型的 Unix 其实没有任何关系。

1977 年, 我第一次使用 Unix, 那时候学习的内容现在仍然管用。如果您 1977 年使用过 Unix, 然后通过时间隧道直接进入 2006 年, 使用运行 Linux 或者 FreeBSD 的计算机, 那么您会有什么感受呢? 您不得不学习使用一些新工具, 包括一个新的、复杂的图形用户界面(Graphical User Interface, GUI)。但是, 您不会感觉到自己来到一个全新的世界。Unix 就是 Unix。

上面是假设, 下面是我的建议。

在许多场合中, 您不必选择必须要使用什么类型的 Unix。例如, 您可能为一家使用商业 Unix——如 AIX(来自 IBM 公司)或者 Solaris(来自 Sun 公司)——的公司工作; 或者您参加了一个课程, 这个课程中的所有人都使用同一类型的 Linux; 又或者您做您祖母的帮手, 而您的祖母坚持使用 FreeBSD, 因为它可以使她想起 70 年代在伯克利分校读书时的所有乐趣。如果就是这种情况, 那么您只需继续使用, 不必担心。Unix 就是 Unix。

如果您正在选择自己的 Unix, 那么下面就是决定的方法:

(1) 如果使用 Unix 是因为您希望学习它如何运转、如何定制工作环境、如何编程或者只是为了寻找乐趣, 那么您可以使用 Linux。

为了帮助您做出选择, 图 2-3 列举了目前最重要的 Linux 发行版。所有这些 Linux 发行版都可以在 Internet 上找到。大多数发行版可以免费下载。如果您无法确定使用哪一种 Linux, 可以使用 Ubuntu。

名称
Debian
Fedora Core
Gentoo
Knoppix
Mandriva(过去叫 Mandrake)
MEPIS
Red Hat
Slackware
SuSE
Ubuntu
Xandros

图 2-3 最重要的 Linux 发行版

(2) FreeBSD 非常稳定和可靠, 并且基本上即装即用。因此, 如果在一家公司工作, 而且正在寻找一种极少需要操心的操作系统, 那么使用 FreeBSD。同理, 如果在家里工作, 并且希望运行一个服务器(如 Web 服务器), 那么使用 FreeBSD。

如果系统不支持 FreeBSD，可以使用 NetBSD。如果对安全非常感兴趣，可以使用 OpenBSD。图 2-4 列举了最常见的 BSD 发行版。

名称
FreeBSD
OpenBSD
NetBSD

图 2-4 最重要的 BSD 发行版

(3) 如果希望在 Microsoft Windows 下运行 Unix，那么可以使用一个免费的产品 Cygwin。一旦安装了 Cygwin，您所需做的全部工作就是打开一个 Cygwin 窗口，在这个窗口中，所有事情看上去都极像 Linux。

我喜欢 Cygwin。尽管我拥有几台 Unix 计算机，但是我使用 Windows 更多些，有了 Cygwin，无论何时我都可以访问喜爱的 Unix 程序。

(4) 最后，如果您希望使用 Unix，但是您希望它运行起来像 Windows，那么可以使用 Macintosh，并运行 OS X。

OS X 是 Macintosh 计算机的操作系统。尽管它拥有类 Mac 的观感，但是它的内部实际上就是 Unix。具体而言，OS X 使用基于 Mach 的微内核、FreeBSD 实用工具以及一个名为 Aqua 的专有 GUI。

为了在 OS X 之下直接访问 Unix，您只需打开一个 Terminal 窗口即可(您可以在 Applications/Utilities 文件夹中找到 Terminal)。

名称含义
OS X
OS X(Macintosh 的操作系统)的发音为“O-S-ten”。这一名称是个双关语。
前一个 Mac 操作系统称为 OS 9，这个操作系统不是基于 Unix 的。因此，“X”既代表罗马数字 10，又可使您联想到 Unix。

2.16 获取 Linux 或者 FreeBSD 的方式

大多数时候，Unix 都是由一张或者多张 CD 安装。为了安装 Linux 或者 FreeBSD，您只需寻找一个免费下载安装文件的站点(这一步骤很容易)，下载文件，刻录 CD，然后使用该 CD 安装操作系统。

还有更好的办法，您可以问问周围的人，看看他们是否拥有这方面的 CD。因为 Linux 和 FreeBSD 都是免费的软件，所以借用 CD 甚至自己制作副本与他人共享都没有任何问题。

当您以这种方式安装 Linux 或者 FreeBSD 时，操作系统就安装在您的硬盘上。如果 Unix 是您的计算机上的唯一操作系统，那么情况比较简单。当您打开计算机时，Unix 就会自动启动。

还有一种情况,即您在计算机上安装了不少一个操作系统。例如,您可能希望同时安装 Windows 和 Unix,从而可以根据需要自由地切换操作系统。但是,您一次只能运行一个操作系统,从一个切换到另一个时还需要重新启动计算机。这样一种设置称为**双重引导系统**。

当使用双重引导系统时,您可以利用一个特殊的程序,这个程序称为**引导加载程序**。每次启动或者重新启动计算机时引导加载程序都会接管计算机。它的任务就是展示一系列可用的操作系统,使您可以选择一种希望启动的操作系统。引导管理器然后将控制转到合适的内核,然后由内核启动操作系统的其余部分(一般而言,如果计算机启动需要花费1分钟时间,那么其中大约有15秒时间花费在引导加载程序上,大约45秒时间花费在操作系统的加载上)。

最常见的 Linux 引导加载程序是 GRUB(Grand Unified Bootloader)和 LILO(Linux Loader)。LILO 已经开发出很长一段时间了,它作为 Linux 发行版的一部分,数年来一直得到很好的维护。GRUB 比较新,是 GNU 的一部分。两个引导加载程序都非常优秀,但是如果要选择的话,建议还是使用 GRUB,这是因为它的功能比较强大,而且更灵活。

为了将机器设置成一个双重引导的系统,您必须将硬盘分成不同部分,这些部分称为**分区(partition)**。为了实现这一目标,可以使用所谓的**分区管理器**。每个操作系统必须使用自己的分区(或者多个分区,如果需要的话)安装。

最常见的双重引导系统中,一个操作系统是 Windows,另一个操作系统是某种类型的 Unix。但是,没有任何理由可以阻止您将硬盘分成多个分区,从而在同一个计算机上安装两个以上的操作系统。例如,您可以在同一个计算机上安装 Windows、Fedora Core Linux 以及 FreeBSD。在这种情况下,引导加载程序将提供3个选择,而不是2个。在这种情况下,您可以说您拥有一个多重引导系统。

第二种运行 Unix 的方式就是不将它安装在硬盘上,您可以从一个称为 Live CD 的特殊 CD 开始启动。

Live CD 指一个可引导的 CD-ROM,包含运行一个完整操作系统所需的所有内容:内核、实用工具等。当从 Live CD 引导时,可以跳过硬盘。这样就允许您随时使用第二个(或者第三个、第四个)操作系统,而不必再在硬盘上安装操作系统。

例如,如果您是一名 Windows 用户,那么通过使用 Live CD,您就可以不必再通过重新分区并安装一个全新的操作系统来体验 Linux 或者 FreeBSD 了。同理,如果您是一名 Linux 用户,那么您可以在不修改系统的情况下使用 FreeBSD。一些人不知道应该使用哪一种类型的 Linux,他们可以使用 Live CD 来尝试各种不同的发行版,寻找适合自己的 Linux 类型。

当使用不属于您的 PC 时,Live CD 也非常有用。例如,如果您在一家强制使用 Windows 的公司工作,那么在没有人看到的情况下,您可以使用 Linux 的 Live CD 来运行 Unix。或者,在为朋友修复计算机时,您可以使用一个拥有您喜爱的工具的操作系统,而不必以任何方式改变朋友的计算机。

为了创建一张 Live CD,您只需选择自己喜欢的系统,然后下载文件,刻录到 CD 中即可。您甚至有可能需要制作好几张 Live CD 来体验不同的系统。

Internet 上有许多不同的 Live CD 可以免费下载。为了方便下载,图 2-5 列举了一些最重

要的 Live CD。如果您不能确定使用哪一种 Live CD, 推荐您使用 Knoppix(发音为“Nop, -pix”)。

提示

如果您的光盘驱动器中有一张 CD, 而系统仍从硬盘启动, 那么请检查 BIOS 设置。您可以修改一个设置项, 从而告诉计算机在从硬盘启动之前先检查是否有可启动 CD。

名称
Damn Small Linux
FreeBSD
Kanofix
Knoppix
MEPIS
PCLinuxOS
SLAX
SuSE
Ubuntu

图 2-5 最重要的 Linux Live CD

在完全硬盘安装和 Live CD 之间您该如何选择呢?

完全安装是一种承诺, 因为它要求对硬盘进行永久修改(当然, 如果您希望的话也可以恢复这些改变)。完全安装的优势在于操作系统永久位于硬盘上。这种方式不仅方便, 而且还允许对系统进行定制, 以及永久地存储文件。

至于 Live CD, 它的承诺比较弱。但是, 除非您为 Live CD 的数据保留了一个特殊的磁盘分区, 否则您无法永久修改或者保存 Unix 数据文件。

另外, 从 Live CD 运行 Unix 将稍微降低计算机的性能。不仅从 CD 引导要比从硬盘引导慢, 而且 Live CD 还必须创建一个 RAM 磁盘来存放通常位于硬盘上的文件, 这需要使用一些内存(RAM 磁盘指用于模拟真实磁盘的一部分内存)。

一种比较出色的折衷方案就是将 Unix 系统安装在一个可移动的存储设备上, 例如 USB 钥匙盘驱动器(key drive, 有时候也叫 USB 闪存驱动器)。这是一种非常小、可移动的设备, 可以插入到 USB 端口充当微型硬盘。

一旦您在钥匙盘驱动器上安装了 Unix, 只需将它插入到一个 USB 端口中, 打开(或者重新启动)计算机, Unix 就会自动启动(您可能需要修改计算机的 BIOS 设置, 从而使计算机明白钥匙盘驱动器在硬盘之前引导)。

通过安装一个钥匙盘驱动器, 您就可以在不修改硬盘的情况下获得永久的 Unix 系统。例如, 当从钥匙盘驱动器启动时, 您可以修改系统, 还可以保存文件, 等等。与 CD-ROM(它是只读的)不同, 钥匙盘驱动器既可读又可写。而且无论何时, 当您希望使用硬盘上的操作系统时, 只需移除钥匙盘驱动器并重新启动计算机即可。

然而, 一旦您决定永久使用一种操作系统, 就会发现将它安装在硬盘上更快捷, 而且更实际。灵活性虽然很好, 但是从长远的观点来看, 和生活中的大多数领域一样, 有承诺

才更出色。

提示

下载并安装任何类型的 Unix 所花费的时间都会超出您的想象,所以在时间紧的情况下不要这样做。

例如,如果您正在度蜜月,当您的妻子更换衣服并化妆时,您只有半个小时的时间待消磨,那么安装一个全新的 Linux 就不是一个好想法。

2.17 什么是 Unix? 什么是 Linux

在结束本章内容之前,我希望总结一下之前的讨论,并且再次回答这个问题:什么是 Unix? 什么是 Linux?

Unix 是一种多用户、多任务处理的操作系统,它由一个类 Unix 内核、许多类 Unix 实用工具以及一个类 Unix shell 构成。Linux 是任何使用 Linux 内核的 Unix 的名称(正如我们所讨论的,术语“类 Unix”没有一个好的定义,当您看到它时知道它即可。)

但是,还有另一种方式来看待 Unix。

多年以来,许多非常、非常聪明的人一直从事 Unix 工作,尽他们的可能创建最好的工具。作为 Unix 用户,您和我都从这些工作中获益。

对我而言,Unix 是一种抽象思想:一种为问题的解决指定特殊方法的实际应用哲理。在使用 Unix 时,您将学习以 Unix 方式处理和解决问题。例如,您将学习如何将简单工具(例如构建块)组合成优美的结构来解决复杂的问题;您还将学习如何自信,从而自学需要知道的大多数内容;而且您还将学习如何以逻辑方式组织自己的思路和行动,从而最好地利用自己的时间和精力。

当您这样做时,您将站在巨人的肩上,而且您的思维也会越变越活跃。现在理解这一点可能非常困难,但是不用担心。只要坚持学习 Unix 一段时间,您就会明白了。

基于这一原因,我先给您另一个 Unix 的定义。据我所知,这是 Unix 最好的定义,而且我希望您在阅读本书时能够记住的正是这个定义。

下面以提示形式给出 Unix 的定义:

提示

Unix 是一组为聪明人准备的工具。

2.18 练习

1. 复习题

1. 什么是操作系统?

2. 什么是内核？列举 4 个由内核执行的任务。
3. 什么是开放源代码软件？它为什么非常重要？
4. 什么是 Linux？第一版 Linux 是在什么时候发行的？是谁发行的？

2. 思考题

1. 如果没有 Unix，那么 Internet 会成为可能吗？如果没有 Internet，那么 Linux 有可能产生吗？

2. 当 Richard Stallman 在 1985 年创立自由软件基金会时，他能够将遍布全球的众多程序员的精力集中起来，一起开发一个自由的操作系统。后来，在 20 世纪 90 年代初期，Linus Torvalds 也能够找到程序员来帮助他开发 Linux 内核。那么，为什么有如此众多的程序员愿意在一个困难的项目上花费大量的时间，而且在完成时又免费向他人提供自己的劳动成果呢？是不是年青的程序员要比年老的程序员在这种情况下更为常见？如果是这样，原因又是什么呢？

3. 传统上，人们希望律师能够作为公共服务提供特定量的免费工作，这样的工作被描述为公共义务服务(pro bono publico，拉丁文为“for the public good”)。编程职业是否对其成员也拥有相似的期望呢？如果有，为什么有呢？如果没有，为什么没有呢？



Unix 连接

Unix 系统中一直都集成有连接不同类型计算机的功能。实际上, Unix 拥有此能力是世界上拥有如此众多计算机网络的主要原因之一(例如, Internet 一直依赖于 Unix 连接)。

在本章中,我们将讨论使 Unix 能够连接到其他计算机(包括 Internet 上和局域网上的计算机)的那些基本概念。通过这种方式,您将学习到这些概念如何支撑所有连接中最基本的 Unix 连接:您和计算机之间的连接。

3.1 人、机器和外星人

我希望首先讨论一个极少明确讨论过的思想。然而,它是一个极其重要的思想,如果您不理解它,那么有许多 Unix 会看上去非常神秘并令人迷惑。这一思想涉及到人类和我们使用机器的方式。

考虑一下生活中最常见的机器如电话、汽车、电视机及收音机等,因为您和机器是分离的实体,所以在使用机器时必须有一种方式使您和机器能够交互。我们称这种功能为界面(interface)。

例如,我们考虑一下手机。该界面包含一组按钮、一个扬声器或者耳机插口、一个小的视频屏幕和一个麦克风。又如汽车的界面包括一把钥匙、方向盘、油门踏板、刹车踏板、多种刻度盘和显示仪表,以及一组杠杆、旋钮和按钮。

这里的观点就是每个人类使用的机器都可以分为两部分:界面和其他部件。

例如,对于桌面计算机而言,界面包括显示器、键盘、鼠标、扬声器和(可能的)麦克风。“其他部件”包括机箱里面的东西:硬盘、CD 驱动器、处理器、内存、视频处理卡以及网络适配器等。

用 Unix 术语来讲,我们称界面为终端(**terminal**)(稍后再解释原因),而将其他部件总称为主机(**host**)。理解这些概念非常关键,因此我准备详细地讨论它们。

因为终端提供界面,所以它有两个主要的任务:接受输入和生成输出。对于桌面计算机而言,输入设施包括键盘、鼠标和麦克风。输出设施包括显示器和扬声器。

为了掌握这些思想,我们可以通过下述两个简单的公式来描述所有的计算机系统:

计算机=终端+主机

终端=输入设施+输出设施

您曾经有过这种想法吗？作为人类，您也由终端和主机构成？换句话说，上述两个公式也可以描述您和我以及其他任何人。

您的“终端”（也就是对世界其他人的界面）提供您的输入设施和输出设施。输入设施包括感觉器官（眼睛、耳朵、嘴、鼻子和皮肤）。输出设施包括能够发音（嘴）及能够改变环境（手、臂、腿以及面部表情肌肉）的身体部分。

什么是您的“主机”呢？相关部件包括：大脑、器官、肌肉以及骨骼、血液、激素等。

将您的“主机”和“终端”分离看上去有人为的意愿，并且还略显滑稽，这是因为您是一个单独的、自我包含的单元。但是考虑一下膝上型电脑，即便所有的部件都是内置的，我们仍然可以分开讨论终端（显示器、键盘、触摸板、扬声器和麦克风）和主机（其他部件）。

假设有两个来自另一个星球的外星人观察您使用膝上型电脑。一个外星人转过来对另一个外星人说：“您看，那儿有一个人在使用他的界面与计算机的界面交互呢。”

对外星人而言，您的界面是否是内置的并不重要，因为膝上型电脑的界面也是内置的。来自另一个星球的外星人看您和我时的视角与我们平时使用的视角并不一样：因为您使用计算机，所以您的界面与计算机的界面通信。实际上，这是使用计算机（或者其他任何机器）的唯一方式。

但是，如果外星人恰好来自一个 Unix 行星又会怎么样呢？在第一个外星人发表了他的评论之后，第二个外星人可能会这样回答：“我明白您的意思。人的终端和计算机的终端如此交互难道不是很有趣的事情吗？”

3.2 价格昂贵的早期计算机

在第 1 章中，我提到最初版本的 Unix 由 Ken Thompson 于 1969 年开发。Ken Thompson 是位于新泽西州的贝尔实验室的一名研究员（当时，贝尔实验室属于 AT&T 公司）。他一直在从事一个大型的、复杂的项目，这个项目就是 Multics，以麻省理工学院为中心。当贝尔实验室决定结束支持 Multics 时，Thompson 返回到新泽西州继续从事他的全职工作，在那里他和其他几位研究人员决定创建他们自己的小型操作系统。特别是 Thompson 有一个他在 Multics 上开发的游戏，这个游戏就是 Space Travel。他希望能够在自己的系统上运行这个程序。

当时，还没有个人计算机。大多数计算机都是大型、昂贵、性能不稳定的机器，要求专用的程序员和管理员（我们称这样的机器为大型计算机）。

大型计算机要求有自己的特殊房间，这样的房间有一个古怪的名称，即“玻璃屋(glass house)”。大型计算机要求玻璃屋的原因有 3 个方面。首先，这种机器比较脆弱，它们需要一个能够控制温度和湿度的环境。其次，这类计算机非常昂贵，通常要花费数百万美元。这样的机器如此之贵，因此不可能允许所有人自由地出入计算机房。玻璃屋可以上锁，从而可以防止除计算机操作员之外的其他人进出。

最后，这类计算机不仅复杂，而且还相当稀有。将这样重要的机器放在玻璃屋内，公

司(或者大学)可以炫耀它们的计算机,特别是对参观者。我还记着当我还是一名年轻的大学生时,我站在 Waterloo 大学巨大的计算机房前面,透过玻璃充满敬畏、胆怯地观看着里面大量的神秘盒子——这些盒子组成了 3 台独立的 IBM 大型计算机。

Multics 运行在一台相似的计算机上,即 GE-645。像大多数大型计算机一样,GE-645 价值昂贵得难以购买、难以租借、难以运行。在那个时代,计算机用户要根据现金数量做好预算,每次运行程序时都要根据运行时间和磁盘存储空间收费。例如,Thompson 每在 GE-645 上运行一次 Space Travel,就要为处理时间花费 75 美元(合 2008 年 445 美元)。

贝尔实验室将 Thompson 和其他研究人员迁回到新泽西州之后,研究人员知道他们再也没有办法接触到另一台大型计算机了。因此,他们开始寻找小的并且易于得到的计算机。

在那个时代,大多数计算机都要花费 100 000 美元,对于个人研究来说这并不是一件简单的事情。但是,1969 年,Thompson 在贝尔实验室搜寻后,发现了一台没有使用的 PDP-7。

PDP-7 由 DEC 公司(Digital Equipment Corporation, 数字装备公司)制造,它是一种所谓的小型计算机(minicomputer)。相对于大型计算机而言,它尺寸比较小、价格比较便宜并且易于得到。按 1965 年美元价值计算,一台 PDP-7 的价格为 72 000 美元,而 GE-645 大型计算机要花费大约 1000 万美元。名称 PDP 是“Programmed Data Processor”的缩写。

这台特殊的 PDP-7 是由一个项目订购的,但是这个项目比较混乱,所以 Thompson 可以霸占它。他编写了许多软件,从而使 Space Travel 能够运行。但是,PDP-7 功能还不够强大,因此 Thompson 和其他几个爱好者开始寻找另外一台计算机。

最终,他们获得了一台较新的 PDP-11。PDP-11 是在 1970 年夏天交付的,其主要优点就是它的基本花费只(!)有 10 800 美元(合 2008 年 64 300 美元)。Thompson 和其他几位研究人员开始使用 PDP-11,在数月之内,他们将 Unix 移植到了这台新计算机上(从图 3-1 中,您可以看到 Thompson 和他的 Unix 伙伴 Dennis Ritchie 在勤奋地工作)。



图 3-1 Ken Thompson、Dennis Ritchie 和 PDP-11

Ken Thompson(就座者)和 Dennis Ritchie(站立者)与贝尔实验室的 PDP-11 小型计算机。Thompson 和 Ritchie 正在使用两个 Teletype 33-ASR 终端将 Unix 移植到 PDP-11 上。

为什么我要告诉您所有这些呢?因为我希望您意识到在 20 世纪 60 年代末期和 70 年代初期,计算机的价格非常昂贵并且不易找到(PDP-11 价格昂贵并且不够用,PDP-7 价格非常昂贵并且更不够用,GE-645 价格非常、非常昂贵并且更加不够用)。因此,对不仅方

便而且便宜的计算机的需求非常巨大。

在使用当前软件的过程中, PDP-11 最大的瓶颈之一就是只能一次只能运行一个程序。这意味着, 每次只能有一个人使用机器。

解决方法就是改变 Unix, 从而使它能够允许一次运行多个程序。但是, 这就没有那么简单了, 直到 1973 年, 这一目标才得到实现——Unix 成为一个完全成熟的多任务处理 (multitasking) 系统 (多任务处理的旧名称叫多道程序设计 (multiprogramming))。

从那时起, 离使 Unix 同时支持不止一个用户就只有一步之遥了, 这一步就可以将 Unix 转换成一个真正的多用户系统 (旧名称叫分时系统 (time-share system))。实际上, 在 1974 年, 当 Thompson 和 Ritchie 发表第一篇描述 Unix 的论文 (参见第 2 章) 时, 他们称之为 “Unix 分时系统”。

但是, 为了进行这样的改变, Unix 开发人员不得不在一个非常重要的概念上达成协议, 即本章前面所讨论的那一个概念: 仅当机器拥有一个合适界面时, 人类才可以使用这台机器。此外, 如果不止一人希望同时使用一台计算机的话, 那么每个人需要一个单独的界面。

这一点非常有意义。例如, 如果两个人希望同时键入命令, 那么计算机不得不连接两个不同的键盘。但是, 在 Unix 的早期, 计算机设备价格昂贵并且难以获得。为了运行一个真正的多用户系统, Thompson 和 Ritchie 从哪里获得这些设备呢?

这个问题的答案具有决定性的意义, 因为它影响了 Unix 的基本设计, 不仅包括早期的 Unix 系统, 而且还包含目前存在的 Unix 系统 (包括 System V、BSD、Linux、FreeBSD 和 OS X)。

3.3 主机和终端

20 世纪 70 年代初期, Ken Thompson 和 Dennis Ritchie 面临一个问题。他们希望将 Unix 转换成一个真正的多任务、多用户操作系统。但是, 这意味着每个用户都需要自己的界面。现在, 高质量的彩色视频显示器、键盘以及鼠标都异常便宜。但是在那个时代, 所有的东西都非常昂贵。计算机系统中没有独立的键盘; 没有鼠标; 就算是计算机的显示器, 价格也昂贵到无法为每个用户都提供一台。

作为一种解决方法, Thompson 和 Ritchie 决定使用一个价格不贵并且可用的机器, 尽管这种机器是为一种完全不同的目的设计的。该机器就是 Teletype ASR33 (ASR 代表自动发送与接收, 即 Automatic Send-Receive)。

最初开发 Teletype 机器的目的是通过电话线发送和接收消息。也正是基于这一点, 该机器被称为电传打字机 (Teletypewriter, 其中 “Teletype” 是一个商标名称)。

最初的实验用电传打字机是在 20 世纪 00 年代初期发明的。在 20 世纪的前半世纪中, 电传打字机这一技术变得越来越完善, Teletype 机器的应用也遍布了全球。AT&T 公司 (贝尔实验室的母公司) 大量参与此类服务。1930 年, AT&T 公司收购了 Teletype 公司。实际上 AT&T 公司这一名称就代表美国电话电报公司。

因此, 在 20 世纪 70 年代早期, 上述情况发生了, Thompson 和 Ritchie 准备使用 Teletype 机器作为他们的新 PDP-11 Unix 系统的界面。大家可以看看图 3-1 中所示的实际机器, 图

3-2 和图 3-3 中给出了 Teletype 的特写镜头。



图 3-2 Teletype 33-ASR

一台 Teletype 33-ASR, 它与 Ken Thompson 和 Dennis Ritchie 在最初的 Unix 系统中使用的相似。

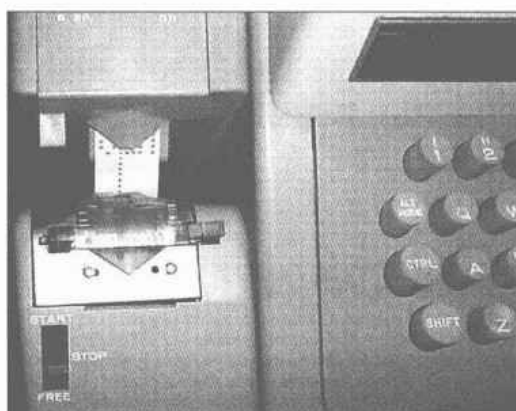


图 3-3 Teletype 33-ASR 特写镜头

Teletype 33-ASR 的特写镜头。请注意高的、圆柱形的按键。为了生成一个字符,必须将键按下半英寸。左边是纸带穿孔机/阅读机。纸带有 1 英寸宽。

作为界面,所有的 Teletype 都有一个键盘用于输入和一大卷纸用于打印输出。为了存储程序和数据,Teletype 中有一个纸带穿孔机(用来在一个长的、窄的纸带上打孔)和一个纸带阅读机(读取纸带上的孔并转换成数据)。

与今天的设备相比,Teletype 相当原始。除了电源之外,所有的东西都是机械的,不是电子的。它没有视频屏幕、没有鼠标,也没有声音。此外,它所提供的键盘也不舒服并且还不容易使用:必须将键按下半英寸深才能生成一个字符(想想这是一种什么样的输入方式)。

使 Teletype 如此珍贵的原因在于它经济实惠并且可用。

下面我们将所有内容集中在一块。Thompson 和 Ritchie 希望创建一个真正的多用户系统。计算设备价格过于昂贵,他们所拥有的就是一些 Teletype 机器(用作界面)和一个单独的 PDP-11 小型计算机(用于处理)。

就像前面提到的外星人一样，Thompson 和 Ritchie 意识到从概念上讲，他们能够将界面与系统的其他部分分离，而这就是他们设计 Unix 的方式。

在这种方式中，处理单元只有一个(他们称它为主机)，界面单元有多个(他们称之为终端)。最初，主机是 PDP-11，终端是 Teletype。但是，这只是为了方便。从原理上讲，Unix 系统可以使用任意数量的主机和任意类型的终端(这样虽是可行的，但是不太实用)。

这一设计决策被证明具有先见之明。从一开始，Unix 为用户和计算机之间达成的连接取决于一个具体的设计原则，而不是具体的硬件。这意味着，多年之后，不管出现了什么类型的新硬件，Unix 的基本组织方式永远都不会改变。

随着终端的日益完美，人们可以将旧的终端拿去，在其位置上插入一个新的终端。随着计算机的日益复杂，功能日益强大，Unix 也需要移植到新的主机上去，同时所有事情都继续按期望的方式工作。

下面将这一思想与微软公司的 Microsoft Windows 相比较。因为 Windows 是为单用户 PC 特别创建的，所以微软公司永远不能彻底地将终端和主机分开。最终的结果是，Windows 没有那么优美、没有那么灵活，而且还永远拘泥于 PC 体系结构。相反，Unix 是优美的、灵活的，而且还适用于任何类型的计算机体系结构。所以，在过了这么多年之后，Unix 的终端/主机模式仍然在不可思议地应用着。

3.4 终端室和终端服务器

前面已经解释过，Unix 是一种多用户系统。这意味着可以有不止一个人同时使用计算机，只要满足以下两个条件：

- (1) 每个人拥有自己的终端；
- (2) 终端与主机连接。

因此，假设一个房间中充满了终端。它们不是计算机，实际上，它们不过就是一个键盘、一个显示器和一些基本的电路。在每个终端的背后是一条穿到地板下面的洞中的电缆，从这里起，一直延伸到看不见的主机计算机。

房间里有许多人。一些人坐在终端前面，输入命令或者盯着他们的屏幕冥思苦想。这些人在同时使用同一台主机计算机。其他人正在耐心地等待轮到他们。这是一个多么繁忙的情景！但可惜的是，没有足够的终端来为每个人提供一台。

我刚才描述的情形就是 20 世纪 70 年代末人们使用 Unix 的情形。在那个时候，计算机——即便是小型计算机——仍然价格昂贵，而且数量也不充分。相对来说，终端价格还不算太昂贵。

因为 Unix 是设计用来支持多用户的，所以经常可以看见终端室内放满了终端，每台终端都连接到主机。当您希望使用计算机时，可以去终端室，等待一台空闲的终端。一旦您找到一台空闲的终端，就可以键入您的用户名和口令来登录系统(我们将在第 4 章中详细

讨论这一过程)。

这一过程概括为图 3-4。

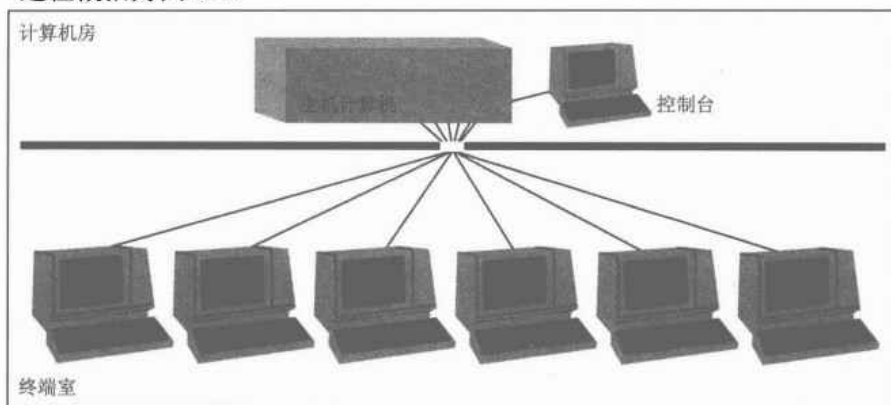


图 3-4 终端室中的终端

在 20 世纪 70 年代末，当计算机仍然昂贵而终端相对便宜时，通常可以看见终端室。在终端室中有多个连接到同一个主机的终端。

一些组织，例如大学的各系或者公司，能负担起多台主机计算机。在这种情况下，允许用户从任意一台终端使用任何一台主机才有意义。为了实现这一目的，需要一台**终端服务器**(terminal server)。终端服务器是一台充当交换机的设备，用于将任意的终端连接到任意的主机。

为了使用终端服务器，您需要输入一条命令告诉终端服务器希望使用哪一台主机。然后终端服务器将您连接到那台主机。接下来您需要以正常的方式输入用户名和口令，并登录系统。

图 3-5 示范了这样一个系统。在这个图中，只给出了 6 台终端和 3 台主机。这或许有点不符合现实。在大型组织中，终端通常有数十台，遍布于整栋建筑物内，连接到多个终端服务器，从而允许访问许多不同的主机。

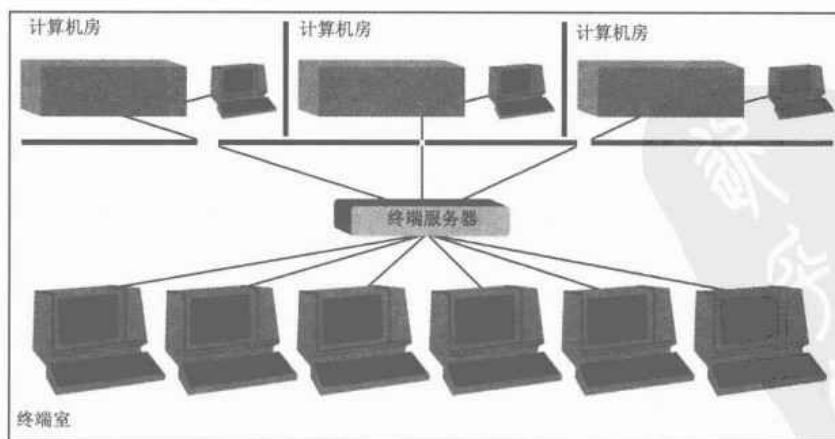


图 3-5 连接到终端服务器的终端

在 20 世纪 70 年代末，一些组织能负担起多台计算机以供用户使用。这种情况下，通常将组织中的所有终端连接到一台充当交换机的终端服务器上，从而允许用户从任意终端访问任意的主机计算机。

3.5 控制台

在所有连接到主机的终端中，有一台终端可能比较特殊。这台终端被认为是计算机本身的一部分，而且它是用来管理系统的。这个特殊的终端就是**控制台(console)**。

为了举一个例子，需要返回到以前那个时代，即回到 20 世纪 70 年代末期。我们去浏览大学的一个系，可以看见一个上锁的房间。在这个房间里面是一台 PDP-11 小型计算机，它边上有一台终端。这台终端直接连接到计算机。这台终端就是控制台，一个特殊的终端，只由系统管理员使用(在前面示范的图 3-4 上就可以看到控制台)。穿过大厅，是一个摆放其他终端的终端室。这些终端是为用户准备的，以使用户远程访问计算机。

现在，让我们直接跳回到当今的时代。您正在使用一台安装有 Linux 的膝上型电脑。尽管 Linux 可以同时支持多个用户，但是您是唯一一位使用该计算机的用户。

您有控制台吗？

是的，您有。因为使用的是 Unix，所以您必须拥有一台终端。在这种情况下，您的终端是内置的：键盘、触摸板、屏幕以及扬声器。该终端也是您的控制台。

一般情况下，控制台由系统管理员使用来管理系统。在第一个例子中，当系统管理员希望使用 PDP-11 的控制台时，他需要进入计算机房，坐在实际控制台前。至于您的膝上型电脑，您就是管理员，而且也只有一台(内置)终端。因此，在任何时候，当您使用您的 Linux 膝上型电脑时，无论是实际管理系统还是只是在工作，您都在使用控制台。

为什么我们需要了解控制台和普通终端呢？原因来自 3 方面。首先，Unix 系统总是区分控制台和普通终端，因此，当您在学习 Unix 的过程中遇到“控制台”时，我希望您知道它的含义。

其次，如果您是一名系统管理员(当您拥有自己的 Unix 系统时就处于这一情况)，有一些特定的事情只能在控制台上完成，不能通过远程终端进行。

(这里举一个例子。如果系统在启动过程中遇到了问题，那么只能通过控制台修复问题。这是因为，直到系统启动以后，您才能通过远程终端访问系统。)

最后，有时候 Unix 系统可能需要显示一个非常严重的错误信息。这样的信息显示在控制台上可以确保系统管理员能够看到它们。

在控制台以及它们为什么如此重要这方面已经说了太多了，下面我希望提一个问题：有没有计算机没有控制台呢？

的确，有许多计算机没有控制台。但是，在解释系统没有控制台时如何工作之前，我需要花点时间讨论一下 Unix 和网络。

3.6 Unix 连接

正如前面所讨论的，Unix 被设计为终端(也就是界面)与主机(处理单元)相分离。这意味着不止一个人可以在同一时间使用同一个 Unix 系统，只要每个人拥有自己的终端即可。

一旦您理解了这一思想，那么询问这样一个问题——即终端要离主机有多远——就很

有意义。答案是根据您的希望，终端可以离主机任意远，但是前提是终端和主机之间必须有一个连接。

当您在膝上型电脑上运行 Unix 时，终端和主机直接连接。当您在桌面计算机上运行 Unix 时，终端通过线缆连接到主机(记住，终端由键盘、显示器、鼠标、扬声器和麦克风构成)。

那么一个更长的距离呢？有没有可能通过局域网(local area network, LAN)将终端连接到主机呢？是的，有这种可能。

例如，假设使用一台连接 LAN 的 PC，在这个 LAN 中有许多台计算机，其中有 3 台计算机是 Unix 主机。使用您的 PC 作为终端访问这 3 台 Unix 主机中的一台是有可能的(当然，在使用任何 Unix 主机之前，必须获得使用这台计算机的授权)。

当使用自己的计算机连接到一台远程 Unix 主机时，您可以运行一个程序，使用您的硬件来仿真(emulate)终端。然后，这个程序通过网络连接到远程主机。

您可以在任意类型的计算机系统上这样做，包括 Windows 计算机、Macintosh 计算机或者另一台 Unix 计算机。一般情况下，终端程序在自己的窗口中运行，而您可以拥有任意数量的独立窗口。

例如，您可以同时打开 3 个窗口，每个窗口运行一个终端程序。每个“终端”可以通过网络连接到一台不同的 Unix 主机。在这种情况下，您将同时在 4 台计算机上工作：您自己的计算机和 3 台 Unix 主机。

图 3-6 描述了这一情形。

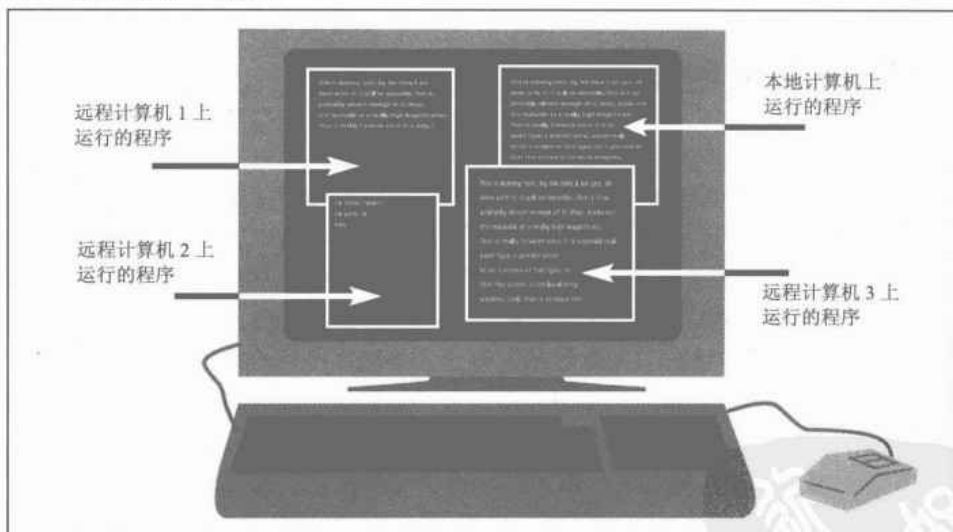


图 3-6 局域网上的 Unix/Linux 计算机

一台局域网上的计算机，运行着 4 个终端程序，每个程序都位于自己的窗口中。其中 3 个“终端”通过网络连接到不同的远程主机。第四个“终端”运行本地计算机上的一个程序。

在图 3-6 中，PC 和 3 台 Unix 主机之间的网络连接和传统网络一样，是通过线缆连接的。但是，任何类型的网络连接都可以实现该功能。特别地，还可以使用无线连接。

下面举一个例子。假设您有 3 位极客朋友：Manny、Moe 和 Jack。而且每个人都有一台运行 Unix 的膝上型电脑。您使用 Debian Linux，Manny 使用 Fedora Core Linux，Moe 使

用 Gentoo Linux, 而 Jack 使用 FreeBSD(Jack 总是有点与众不同)。

你们聚在一起召开一个 Unix 集会(也就是说, 计算机、含咖啡因的饮料和垃圾食品), 大家决定每个人都应该访问其他 3 台计算机。首先, 每个人都在自己的计算机上为其他三人创建账户(这里不再详细说明具体步骤, 不过过程并不太难)。

然后, 所有人使用 `iwconfig` 命令或者 `wiconfig` 命令配置自己的计算机, 从而允许其他人无线连接到一个小型网络。

一旦这个网络建立起来, 你们每个人都在自己的计算机上打开 3 个终端窗口。在每个窗口内, 你们都连接到其他 3 台计算机中的一台。

现在房间内有 4 个人, 每个人都在自己的膝上型电脑上运行一种不同类型的 Unix, 每台计算机又访问着另外 3 台计算机。还有什么事情比这更酷吗?

到现在为止, 我们已经讨论了直接连接到主机的终端(膝上型电脑)、通过线缆连接到主机的终端(桌面计算机)、通过一般 LAN 连接到主机的终端以及通过无线 LAN 连接到主机的终端。我们是不是可以继续向下进行了?

是的。通过使用 Internet 将终端连接到主机, 我们就拥有了一个可以到达全球任何地点的连接。实际上, 我经常使用网络从我的 PC 连接到远程 Unix 主机。为了这样做, 我需要打开一个终端窗口, 并连接到远程主机。只要拥有一个好的连接, 我感觉上就像在使用一台位于大厅的计算机一样。

3.7 没有控制台的主机

前面提到过世界上有许多没有连接终端的 Unix 主机计算机。这是因为, 如果计算机独自运行, 不需要人类的直接输入, 那么计算机就不需要终端。这样的计算机称为无头系统(headless system)。

在 Internet 上, 有许多运行的 Unix 主机没有终端。例如, 有数百万个无头系统充当了 Web 服务器和电子邮件服务器, 默默地做着自己的工作而不需要人类的干预。这些服务器大多数都运行着 Unix, 并且它们大多数都没有连接终端。

(Web 服务器响应网页的请求, 并向外发送合适的数据。电子邮件服务器发送和接收电子邮件。)

如果需要直接控制这样的主机计算机——例如需要配置机器或者解决问题——那么系统管理员只需简单地通过网络连接到主机。当系统管理员完成工作后, 他只需断开主机的连接, 让主机接着独自运行。

在 Internet 上, 有两种非常常见的主机类型, 它们自动运行, 不需要终端。第一种是服务器, 例如前面提到的 Web 服务器和电子邮件服务器。稍后我们再讨论它们。

第二种是**路由器**: 一种特殊用途的计算机, 将数据由一个网络中继到另一个网络。在 Internet 上, 路由器提供网络之间真正的连接。

例如, 当您发送一个电子邮件消息时, 数据在到达目的计算机的路途上要通过一系列的路由器。这是自动进行的, 不需要人的任何干预。全世界有数百万台路由器, 每天 24 小时自动地工作, 它们中有许多是没有控制台的 Unix 主机。

如果路由器出现问题该怎么办呢？在这种情况下，这是系统管理员的工作。他在自己的 PC 上打开一个终端窗口，连接到路由器，修复问题，然后再断开与路由器的连接。

一些拥有许多 Unix 服务器的大型公司使用一种不同的方法。它们将每个主机计算机的控制台连接到一台特殊的终端服务器上。通过这种方式，当出现问题时，系统管理员可以使用终端服务器直接登录到有问题的计算机。我有一位朋友，他曾经在一个拥有 95 台不同 Unix 主机的公司工作过，这些主机连接到一组终端服务器上，而这些终端服务器只是为了系统管理使用。

3.8 客户端/服务器关系

在计算机术语中，提供某种类型的服务的程序称为**服务器**，使用服务的程序称为**客户端**。

当然，这些术语来自商业领域。如果您去见律师或者会计师，您就是客户，而他们为您服务。

客户端/服务器关系是一个基本的概念，在网络和操作系统中都使用。客户端和服务端不仅在 Internet 上广泛使用，而且也是 Unix(和 Microsoft Windows，就此而言)的重要部分。考虑下面的例子。

我确信您一定知道，为了访问 Web 您需要一个叫做浏览器的程序(最重要的两个浏览器分别是 Internet Explorer 和 Firefox。Internet Explorer 的应用最广，但是 Firefox 最好)。

假设您决定访问我的网站 <http://www.harley.com>。首先，您在浏览器的地址栏中输入地址，然后按回车键。浏览器然后向 Web 服务器发送一个消息(这里跳过了几个细节)。在接收到请求后，Web 服务器通过向您的浏览器发送数据进行响应。浏览器然后以 Web 页面的形式显示数据(在这个例子中，就是我的主页)。

刚才描述的就是一个客户端/服务器关系。客户端(浏览器)代表您联系服务器。服务器发送回数据，然后客户端处理数据。

下面再举一个例子。电子邮件的使用有两种方式。您可以使用基于 Web 的服务(例如 Gmail)，也可以在您的计算机上运行一个程序来代表自己发送和接收邮件。我准备讨论一下第二种类型的服务。

当您运行自己的电子邮件程序时，它使用不同的系统来发送和接收邮件。为了发送邮件，它使用 SMTP(Simple Mail Transport Protocol，简单邮件传输协议)。为了接收邮件，它或者使用 POP(Post Office Protocol，邮局协议)，或者使用 IMAP(Internet Message Access Protocol，Internet 消息访问协议)。

假设您刚刚写完一封电子邮件，而且您的电子邮件程序已经准备发送这封电子邮件。为了发送邮件，电子邮件程序临时成为一个 SMTP 客户端并连接到一个 SMTP 服务器。然后，您的 SMTP 客户端请求 SMTP 服务器接收消息，并向其发送消息。

同样，当您查看发送过来的邮件时，您的电子邮件程序临时变成一个 POP(或者 IMAP)客户端，并连接到 POP(或者 IMAP)服务器。然后，它询问服务器是否有发送过来的邮件。如果有，服务器就将邮件发送给您的客户端，客户端接着采取合适的方式处理消息。

我猜即使您已经发送和接收电子邮件有数年时间，也可能从来没听说过 SMTP、POP

和 IMAP 客户端和服务端。同样，您可能已经使用 Web 有数年时间，但还不知道浏览器实际上就是一个 Web 客户端。这是因为客户端/服务器系统通常工作得非常出色，以至于客户端和服务端能够独自完成工作，而不必在细节上打扰用户(您)。

一旦知道了 Unix 和 Internet，您就会发现到处都是客户端和服务端。下面我们举 3 个这方面的例子。

第一个例子，为了连接一台远程主机，您使用一个叫 SSH(SSH 代表 secure shell，即安全 shell)的客户端/服务器系统。在使用 SSH 时，您在自己的终端上运行一个 SSH 客户端，然后由 SSH 客户端连接位于主机上的 SSH 服务器。第二个例子，为了从远程计算机上下载或者向远程计算机上传文件，您需要使用一个叫 FTP(File Transfer Protocol，文件传输协议)的系统。在使用 FTP 时，您在自己的计算机上运行一个 FTP 客户端。然后，FTP 客户端连接 FTP 服务器。接下来，客户端和服务端一起工作，根据您的愿望传送数据。

当您成为一名有经验的 Unix 或者 Linux 用户时，您将会发现自己使用过这些系统。此时，您才会意识到客户端/服务器模型的美丽和强大。

最后一个例子，您可能听说过 Usenet，这是一个世界范围内的讨论组系统(如果没有听说过，您可以访问网站 <http://www.harley.com/usenet> 去了解一下)。为了访问 Usenet，您需要运行一个 Usenet 客户端，这个客户端叫 newsreader。然后，newsreader 连接叫做 news server 的 Usenet 服务器(稍后再解释这些名称)。

所有这些例子都是不同的，但是有一件事情是相同的。那就是在每一个例子中，客户端从服务器请求服务。

严格地说，客户端和服务端都是程序，不是机器。但是，非正式地讲，术语“服务器”有时候指运行服务器程序的计算机。

例如，假设您去一家公司参观，向导向您展示一个里面有两台计算机的房间。他指着左边的计算机说：“那是我们的 Web 服务器。”然后，他指着另一台计算机说：“那是我们的邮件服务器。”

名称含义

newsreader、news server

全世界范围讨论组的 Usenet 系统由两名 Duke 大学的研究生在 1979 年启动，他们是 Jim Ellis 和 Tom Truscott。Ellis 和 Truscott 设想将 Usenet 作为北卡罗来纳州两所大学(North Carolina 大学和 Duke 大学)发送新闻和声明的一种方式。

在极短的时间内，Usenet 扩散到其他学校。在几年之内，它就发展成为一个庞大的讨论组系统。

因为它的起源，所以 Usenet 仍然指新闻(有时候也叫网络新闻)，即使它不是一个新闻服务。同样，讨论组被称为新闻组，客户端称为 newsreader，而服务器被称为 news server。

3.9 按下键时发生的事情

现在您已经明白，Unix 是基于终端和主机这一思想的。其中，终端充当界面，主机进

行处理。

终端和主机可能是同一台计算机的不同部分，例如膝上型电脑或者桌面计算机。终端和主机二者之间也可能彼此完全分离，如通过 LAN 或 Internet 访问 Unix 主机。

无论如何，终端/主机关系是一个深深嵌入到 Unix 结构中的关系。前面已经说过，我希望提一个看上去很简单的问题：“当您按下键时会发生什么事情呢？”答案可能要比您想象的复杂得多。假设您使用一台 Unix 计算机，且您希望知道现在的时间。显示时间和日期的 Unix 命令是 `date`。因此，您按下 4 个键 `<d>`、`<a>`、`<t>`、`<e>`，然后又按下 `<Enter>` 键。

随着您按下键，每个字母都会显示在屏幕上，因此可以很自然地猜测您的终端正在显示您键入的字母。实际上，情况并非如此。是主机，而不是终端负责将刚才键入的内容显示出来。

每次按下键时，终端都向主机发送一个信号。然后主机作出响应，在屏幕上显示合适的字符。

例如，当按下 `<d>` 键时，终端向主机发送一个含义为“用户刚才发送了一个 `d` 字符”的信号。然后主机发送回一个含义为“在终端屏幕上显示字母 `d`”的信号。当这种情况发生时，我们称主机将字符回显(echo)在屏幕上。

当您使用鼠标时也会发生相同的事情。移动鼠标或者单击鼠标按键也会向主机发送信号。主机解释这些信号，并向终端发送回指令。然后终端在屏幕上进行合适的变化：移动指针、调整窗口大小、显示菜单等。

在大多数情况下，所有事情发生得如此之快，就好像键盘和鼠标直接连接到屏幕一样。但是，如果使用一个长距离的连接，例如通过 Internet，那么您有时候会注意到按键时间和看到字符显示在屏幕上的时间之间的延迟。当移动鼠标或者按鼠标按键，而屏幕没有立即更新时，也意味着发生了延迟。我们称这一延迟为滞后(lag)。

您或许会问，为什么 Unix 要设计成主机回显每个字符呢？为什么主机不会默默地接受它所接收的内容，而让终端处理回显呢？这样做将比较快速，从而避免滞后。

答案就是当主机进行回显时，您可以看到键入的内容被成功地接收，而且终端和主机之间的连接完整无缺。如果终端进行回显，那么当遇到问题时，您就不知道到主机的连接是否正常。当然，当您使用一台与主机物理分离的终端时这一点尤为重要。

抛开可靠性不说，Unix 设计人员选择让主机进行回显还有另外一个原因。正如我将在第 7 章讨论的，有一些特定的键(例如 `<Backspace>` 或者 `<Delete>`)，按下这些键可以修改刚才键入的内容。Unix 的设计目的是要与许多不同的终端一起工作，因此操作系统本身以统一的方式处理按键才有意义，而不要让每种不同类型的终端以自己的方式处理按键。

提示

当使用 Unix 时，您键入的字符通过主机回显到屏幕上，而不是通过终端。大多数时候，滞后量非常小，以至于您注意不到。但是，如果通过一个低速连接使用远程主机，那么您键入的字符显示在屏幕上之前存在一个延迟的情况会时有发生。

Unix 允许提前键入许多字符，因此不用担心，您只需不停地键入，最终主机将捕获所有的输入。在几乎所有情况下，无论滞后有多长，内容都是不会有任何丢失的。

3.10 字符终端和图形终端

广义地讲, Unix 中可以使用的终端有两种类型。如何与 Unix 交互取决于您使用的是哪一种类型的终端。

请大家再花一点时间看看图 3-2 和图 3-3, 即 Teletype ASR33 的照片。正如前面所讨论的, 该机器是最早的 Unix 终端。如果仔细地观察它, 您就会看到该机器唯一的输入设备是一个未成熟的键盘, 而唯一的输出设备是一卷纸, 字符就打印在这些纸上。

多年以来, 随着硬件的发展, Unix 终端也越来越先进。键盘越来越完善并且更方便易用, 而卷纸则被带有视频屏幕的显示器所替代。

然而, 在很长一段时间内, Unix 终端有一个基本特征没有改变: 输入和输出只有一种形式, 即字符(也称为文本)。换句话说, 就是只有字母、数字、标点符号和几个特殊的控制键, 但是没有图形。

一个只使用文本的终端被称为字符终端(character terminal)或者基于文本的终端(text-based terminal)。随着 PC 技术的发展, 出现了一种新类型的终端, 即图形终端(graphics terminal)。图形终端在输入方面仍依靠键盘和鼠标, 而在输出方面, 它充分利用了视频硬件的优点。图形终端不仅可以处理文本, 而且还可以显示任何可以使用小点在屏幕上进行绘制的内容: 图形、几何形状、阴影、线条、彩色等。

很明显, 图形终端的功能比字符终端强大许多。当使用字符终端时, 只限于键入字符和阅读字符。而使用图形终端时, 可以使用一个成熟的 GUI(graphical user interface, 图形用户界面), 包括图标、窗口、颜色、图形等。

基于这一原因, 您可能认为图形终端永远比字符终端好。毕竟, 难道 GUI 不是总比纯文本好吗?

对于使用 Microsoft Windows 的 PC 和 Macintosh 来说确实如此。从一开始, Windows 和 Macintosh 操作系统的设计就基于 GUI。实际上, 它们依赖于 GUI。

Unix 就不同了。

因为 Unix 是在字符终端时代开发的, 操作系统的所有能力和功能利用的都是纯文本。尽管也有 Unix 的 GUI(将在第 5 章中讨论), 并且您确实需要学习如何使用它们, 但是 Unix 的大量工作——包括我在本书中讲授的所有内容——只要求纯文本。对于 Unix 来说, 图形很漂亮, 但并不必要。

这在实际过程中有什么含义呢? 当您在自己的计算机上使用 Unix 时, 您可以使用 GUI(使用鼠标、操纵窗口等)。这意味着您的计算机将仿真一个图形终端。

但是, 大多数时间, 您会发现自己在一个充当字符终端的窗口中工作。在这个窗口中, 您所键入的都是文本, 所看到的也都是文本。换句话说, 就是您正在使用一个字符终端。以相同的方式, 当您连接到一台远程主机时, 您通常通过打开一个充当字符终端的窗口来处理工作。

当您发现自己以这种方式工作时, 我希望您花一点时间思考一下: 您正在以和 20 世纪 70 年代那些最初的 Unix 用户一样的方式使用 Unix。多么有趣的事情! 30 多年过去了, 系统仍然出色地工作。大多数时候, 文本就是您的全部需求。

3.11 最常见类型的终端

多年以来, Unix 被开发为和差不多数百个不同类型的终端一起工作。当然, 现在我们已经不再使用单独的硬件终端: 我们使用计算机来仿真终端。

前面曾经提过打开一个窗口仿真字符终端的思想。在大多数情况下, 仿真基于一种非常古老的终端的特征, 这个终端就是 VT100, 可以追溯到 1978 年(VT100 由 DEC 公司生产, 和在本章开头讨论的 PDP-11 计算机的生产厂家相同)。尽管实际的 VT100 已经有数年没有使用了, 但是它们的设计非常优良, 并且曾经非常流行, 它们为字符终端设立了一个永久的标准(图 3-7 中显示了一台实际的 VT100)。



图 3-7 VT100 终端

最流行的 Unix 终端 VT100, 由 DEC 公司于 1978 年提出。VT100 是如此的流行, 以至于它被设立为一个永久的标准。即便是现在, 大多数终端仿真程序都使用基于 VT100 的规范。

当然, 图形终端拥有不同的标准。正如您即将看到的(第 5 章中), Unix 的 GUI 都基于一个称为 X Window 的系统, 而且对 X Window 的基本支持由一个称为 X 终端的图形终端提供。今天, X 终端是图形终端仿真的基础, 就如同 VT100 是字符终端仿真的基础一样。

因此, 当您连接到一台远程主机时, 您有两个选择。可以使用一个字符终端(最常见的选择), 在这种情况下仿真的是一个 VT100 终端或者类似于 VT100 的终端。或者, 如果您希望使用 GUI, 那么可以使用一个图形终端, 这时您仿真的一个 X 终端。

尽管现在不会太过深入地详细解释, 但是我将向您示范两个将要使用的命令。为了连接一台远程主机并仿真一个字符终端, 您可以使用 `ssh`(secure shell)命令。为了仿真一个 X Window 图形终端, 可以使用 `ssh -X` 命令。

3.12 练习

1. 复习题

1. 最早的 Unix 终端使用什么类型的机器，为什么要选择这一类型的机器？
2. 什么是终端室？为什么它们有必要存在？
3. 什么是无头系统？列举两个在 Internet 上使用的，用来提供非常重要服务的无头系统的例子。
4. 什么是服务器？什么是客户端？

2. 思考题

1. 1969 年，贝尔实验室的 Ken Thompson 正在寻找一台计算机，来创建第一个 Unix 系统。他发现了一台可以使用的 PDP-7 小型计算机。假如 Thompson 没有找到 PDP-7，我们今天还会有 Unix 吗？

2. 在 20 世纪 70 年代，计算机(即便是小型计算机)的价格都非常昂贵。因为没有人拥有自己的个人计算机，所以人们必须共享计算机。相比今天，20 世纪 70 年代的计算机速度缓慢，而且内存和存储设施有限。现在，每名程序员都拥有自己的计算机，并且这些计算机都拥有快速的处理器、大量的内存、众多的磁盘空间、完善的工具以及 Internet 接入。那么谁更有乐趣呢？是 20 世纪 70 年代的程序员，还是现在的程序员呢？从现在起 20 年后的程序员又如何呢？



开始使用 Unix

当您参加关于如何使用 Unix 的初期课程时，需要学习的内容依赖于您访问 Unix 的方式。您是作为一个共享的多用户系统的一部分(例如在学校或者单位中)? 还是拥有一台自己的 Unix 计算机(在这种情况下，您控制计算机并且是唯一的用户)?

在本章中，我们将讨论第一种情形：这种情况下您使用的 Unix 系统由其他人维护。我将示范如何启动和停止一个工作会话，而且还将解释一些基本的概念，例如系统管理员、口令、用户标识、用户和超级用户。在向您解释这些思想的过程中，将假定您使用一个简单明了的基于文本的界面。

在第 5 章中，我们将讨论较复杂的情形。在这种情形中，您使用安装在自己计算机上的属于自己的 Unix 系统。在那一章中，我们还将讨论一些图形界面的思想。

如果您不准备将 Unix 作为共享系统使用，又或者您只希望使用自己的计算机并且只使用图形界面，那么您还需要阅读本章内容吗?

答案是肯定的。无论怎样使用 Unix，本章中涵盖的技能和思想都是 Unix 的基础，对每个人都同样重要。

4.1 系统管理员

从广义上讲，Unix 系统的访问有两种方式。第一种是拥有自己的 Unix 计算机，在这种情况下，您是唯一的用户，负责管理系统。

另外一种是使用一个共享的多用户系统——在学校或者在单位——在这种情况下，您只是用户之一。如果是这种情况，那么将由其他人负责管理系统，因此您不用担心系统的维护。但是，此时您必须遵循一些规则，在一些限制内进行工作。

当然，在访问 Unix 时这两种方式都可以采用。例如，您可以在学校里使用一个共享的系统，而在家时使用自己的 PC。

尽管拥有自己的 Unix 计算机听起来相对简单些，但是实际情况并非如此。真实情况是使用一个共享的系统更简单些。因为您不拥有系统，所以有其他人为您管理系统，而这是一个相当重要的任务。

所有的 Unix 系统都需要管理和维护。执行这些职责的人称为系统管理员(system

administrator), 通常简称为 sysadmin 或者 admin(旧术语为 system manager, 但是现在已经不再使用)。

如果您使用的计算机由某个组织(例如大学或者公司)所拥有, 那么系统管理员可能是一名雇佣的员工。实际上, 在拥有 Unix 计算机网络的组织内, 系统管理是一份全职的工作, 需要大量的专业知识。系统管理员可能还有许多助手。

在 20 世纪 90 年代中期以前, 拥有自己的 Unix 计算机通常并不多见。大多数使用 Unix 的人都是在访问一个共享的系统。Unix 系统由组织(通常是学校或者公司)负责维护, 他们还制定了一些规则要求所有用户遵守。

大多数时候, 人们或者通过一个终端, 或者通过一台 PC 机仿真一个终端(参见第 3 章)来远程访问 Unix。就这一点而论, 使用 Unix 最常见的方式就是基于文本界面, 即只使用一个键盘和一个显示器(就像我们在本章中所做的一样)。只有极少数的人才使用图形界面的 Unix。

当您拥有自己的个人 Unix 计算机时, 不论好坏, 您自己就是系统管理员。在好的情况下, 您管理自己的系统, 这是一项具有极高实践价值的活动, 可以增强您的技能和信心, 使您感觉到真正地控制着自己的计算机。在坏的情况下, 您有时候会感觉到挫折和迷惑。

为了很好地使用 Unix, 您需要理解一些基本的概念: 文件系统、文本编辑器、shell 等。所有这些我都在本书中讲授。为了成为一名大型系统或者网络的高效管理员, 您需要学习更多的知识。您不得不掌握大量深奥的技能, 其中许多已经超出了本书的范围。相比之下, 管理自己的个人系统要简单许多。您所需知道的全部内容就是基本的 Unix 技能和一种细心的态度。

无论如何, 不管您花费了多长时间来学习如何熟练地管理自己的 Unix 计算机, 我可以向您保证, 系统管理永远是一个经验学习的过程(如果没有学会别的, 那么至少学会了有耐心)。

下面让我们看看当其他人为我们管理系统时情况是怎么样的。

4.2 用户标识和口令

在使用 Unix 计算机之前, 系统管理员必须给您一个用来向系统标识自己的名称。这个名称称为用户标识(userid)。单词 userid 是“user identification”的缩写, 它的发音为“user-eye-dee”。

除用户标识之外, 您还将得到一个口令(password), 每次启动工作会话时, 都必须键入口令。

一旦拥有使用 Unix 系统的权限, 我们就可以说您拥有了这台计算机的账户。即使您没有真正为账户付钱, Unix 也会记录您使用系统的情况(Unix 提供了许多内置的账户管理功能, 系统管理员可以使用它们记录各个账户的操作)。另外, 您的账户可能有一些特定的预定义限制, 例如允许您的文件使用多大的磁盘空间, 或者您可以打印多少页内容等。

如果您是一名学生, 极有可能遇到的一个限制就是您的账户的截止时间。例如, 您的账户可能在学期末自动终止, 这样才合理。

用户标识是怎样确定的呢? 大多数情况下, 系统管理员将为您选择一个用户标识。一种常见的方法就是基于个人的真实姓名来确定用户标识。例如, 对于 Harley Q. Hahn 来说,

用户标识可能是 harley、hahn、hhahn、harleyh 或者 hqh。

另外，用户标识还可能反映一些客观标准。例如，如果您是一名学生，您是班级 CS110 中第 25 位申请 Unix 账户的人，那么就有可能为您分配一个用户标识 cs110-25。

每当启动一个 Unix 会话时，必须输入自己的用户标识。从这时起，Unix 就使用这个名称来标识您。例如，当您创建文件时，它们不属于您；它们将被您的用户标识所“拥有”（本章稍后将讨论这一区别）。

用户标识不是保密的，理解这一点非常重要。例如，如果使用 Unix 发送电子邮件，那么您的用户标识将是电子邮件地址的一部分。此外，在任何时候，任何人都可以方便地显示所有当前正在使用系统的用户标识，而且——如果您知道自己正在做什么的话——您还可以显示系统中注册的所有用户标识。

当然，安全非常重要，但是这并不要求用户标识是秘密的。较合理的方式是通过确保口令是秘密的来维护安全。通过这种方式，每个人都可以查看谁在使用计算机，但是对系统的访问是受控制的。

口令或许是一组没有意义的字符，例如 H!lg%12，它也是一组其他人很难猜测的字符。本章稍后将解释如何改变口令，以及什么类型的口令才适合于使用。

4.3 登录(开始使用 Unix)

当您坐在终端前面，启动工作的这一过程称为**登录**。尽管这一思想很简单，但是这个术语有点棘手。

当我们以动词讨论这一思想时，写成两个单词：“log in”。当我们以名词或形容词讨论这一思想时，写成一个单词 login。

例如，“In order to log in, you need to learn the login procedure(为了登录系统，您需要学习登录过程)”或者 “We need a larger computer. Our current one gets over 500 logins a day; there are too many people trying to log in at the same time(我们需要一个更大的计算机。我们现在一天有 500 多次登录，同时还有许多人在试图登录)。”

实际的登录过程相当简单。您所需做的全部事情就是键入您的用户标识和口令。下面介绍它的工作方式。

当 Unix 程序希望您键入一些东西时，它就显示一个提示(prompt)，即一个指示必须从键盘输入一些内容的短消息。当 Unix 希望向您表示它正在等待您登录时，它显示下述提示：

```
login:
```

Unix 系统这样说：“Type your userid and press the <Return> key(输入用户标识并按下 <Return>键)。”

尽管这看上去非常简单，但是我还是希望暂停一会，先回答一个重要的问题，即准确地说，<Return>键指什么呢？

在第 7 章中，您将了解到 Unix 使用一组特殊的键，这些键并没有必要对应到每个键盘上相应的物理键。那时，我们再详细地讨论细节。现在，我希望您知道的全部内容就是

Unix 有一个特殊的键，当按下这个键时就表示您已经结束一行内容的输入。这个键就称为 `<Return>` 键。当按下 `<Return>` 键时，它就向 Unix 发送一个称为新行(newline)的信号。

如果您的键盘上有实际的 `<Return>` 键，那么按下它将发送一个新行信号(Macintosh 机采用这种方式)。否则，您需要通过按下 `<Enter>` 键发送新行信号(PC 机采用这种方式)。因此，当本书中告诉您按 `<Return>` 键时，您需要或者使用 `<Return>` 键，或者使用 `<Enter>` 键，这要看您的键盘上拥有哪一个键。

一旦键入了您的用户标识并且按下了 `<Return>` 键，Unix 就会显示下述提示来询问您的口令：

```
Password:
```

在键入口令的过程中，您会发现口令并不回显。这是为了防止其他人碰巧从您的肩膀上看过过去观察到您的口令(记住，在 Unix 中，用户标识并不保密，但是口令需要保密，这就是为什么在登录过程中，当输入用户标识时回显，而输入口令时不回显的原因)。

还要注意，Unix 与 Windows 不同，当您键入口令时，系统并不为每个字符显示一个星号。这意味着如果某人在看您的屏幕，他不但看不到您的口令，而且还不知道您键入了多少个字符。

在键入口令之后，再次按 `<Return>` 键。Unix 将检查并确认您的口令是否有效。如果口令有效，那么 Unix 将完成登录过程，您就可以开始工作了。

如果您的用户标识或者口令不正确，那么 Unix 将显示以下信息并让您再次尝试：

```
Login incorrect
```

如果您是远程连接，而且不正确登录的次数太多，一些系统将断开连接。这就使那些试图通过不停地猜测口令来潜入系统的人无法成功(一般情况下，您可以尝试 3~5 次，准确的次数由系统管理员控制)。

在键入用户标识和口令时，有 3 件重要的事情希望您能记住。

- 一定不要将小写字母和大写字母混淆。例如，如果您的用户标识是“harley”，您就要键入 **harley**，而不是 **Harley**。
- 小心不要将数字 0 和大写字母 O 混淆。
- 小心不要将数字 1 和小写字母 l 混淆。

在结束本节内容之前，我希望指出一个很少有人注意到的奇怪事情。在几乎所有的 Unix 系统上，登录程序都显示 **login:**(第一个字母是小写的“l”)和 **Password:**(第一个字母是大写的“P”)。没有人知道为什么。

提示

当键入用户标识时，Unix 总是询问口令，即使这个特定的用户标识是无效的。这就使怀有恶意的人难以猜测用户标识。

例如，如果有人输入用户标识 **harley**，Unix 系统总会向他询问口令，即使系统中没有注册这个用户标识。

当然，这还意味着如果错误地输入了用户标识或者口令，那么您也不知道是哪一个错了，Unix 系统只会告诉您登录错误。

名称含义

<Return>键

现在, 大多数键盘只拥有<Enter>键, 而不再拥有<Return>键。那么为什么, 以及哪个时候 Unix 开始使用名称<Return>呢?

这一切要从传统说起。正如第3章中解释的, 多年以来, Unix 一直通过终端访问, 而不是通过 PC, 碰巧的是所有的终端都有一个<Return>键。尽管现在有无数的 PC 键盘提供<Enter>键, 但是 Unix 术语没有改变。

名称“Return”来源于电传打字机。在旧年代, 机械电传打字机中存放纸张的部件称为“托架”。每放入一张新纸时, 托架就从最右边开始。在键入字符时, 托架每一次向左移动一个字符。

当您到达一行的末尾时, 需要使用您的左手推动操作杆将托架移回右边。同时, 操作杆还将纸张向上移动一行。通过这种方式, 纸张就定位到新行的起始位置上。

第一种 Unix 终端是 Teletype ASR33 机器(参见第3章)。与电传打字机不同, 它们没有可移动的托架。但是, 当打印文本时, 从一行的尾部转到下一行的开头依然包含两个单独的运动。这两个运动与推动电传打字机上的操作杆时发生的情况相似, 因此人们使用电传打字机的术语来描述它们, 称之为托架回车(carriage return)和换行(linefeed)。这两个术语在 Unix 世界中仍然重要, 您将不时地遇到它们。

图 4-1 给出了一张 Teletype ASR33 键盘的照片。注意, Teletype ASR33 键盘上既有<Linefeed>键, 也有<Return>键。这就是为什么时至今日 Unix 仍然称终止一行文本的键为<Return>键的原因。



图 4-1 Teletype ASR33 的键盘

Teletype ASR33 的键盘, 最早用作 Unix 终端的设备。注意键盘上的<Linefeed>键和<Return>键(在第二排的最右边)。

4.4 登录之后发生的事情

在成功登录之后, Unix 将显示一些信息, 后面跟一个邀请, 让您输入命令。然后, 您就可以通过输入一条接一条的命令来启动工作会话。

登录成功后显示的信息各不相同, 这取决于系统管理员对系统的配置。例如, 图 4-2

示范了一个典型的 FreeBSD 系统显示的信息。

```
Last login: Sat Jun 28 17:02:18 on ttyt1

Copyright 1980, 1983, 1986, 1988, 1990, 1991, 1993, 1994
The Regents of the University of California.
All rights reserved.

FreeBSD 4.9-RELEASE: Wed Mar 8 16:26:07 PDT 2006

Welcome to FreeBSD!
```

图 4-2 登录消息

在成功登录系统之后，将会看到欢迎消息。在这个例子中，看到的是 FreeBSD 系统显示的典型消息。

第一行显示当前用户标识上一次登录该计算机的时间。在这一行的末尾，我们看到了 `ttyt1`，它是当时所使用终端的名称。

无论何时，当您登录系统时，都要花一些时间检查这一行。这样做是基于安全考虑。如果您看到的时间比您记忆中的时间更近，那么就可能有人在没有经过您的授权的情况下使用了您的账户。如果这样，您需要立即改变口令(本章稍后将解释如何改变口令)。

接下来的 3 行包含的是版权信息。第 3 章中讲过，FreeBSD 基于加利福尼亚大学伯克利分校的工作，因此将加利福尼亚大学作为 FreeBSD 的版权所有者是可以理解的。

倒数第二行显示我们正在使用版本 4.9 的 FreeBSD。日期和时间显示内核的“构建”时间——也就是生成时间——为 2006 年 3 月 8 日下午 4:26(Unix 使用 24 小时制时钟)。

最后一行是友好的 FreeBSD 程序员对大家的问候。

在登录信息显示之后所发生的事情部分取决于系统的设置。作为登录过程的一部分，Unix 执行一系列存放在特殊的初始化文件中的预定义命令。其中一些文件受系统管理员控制，因此他可以确保每次有人登录时都能够执行特定的任务。例如，系统管理员可能希望，无论何时，当用户登录时都向用户显示一个特定的消息。

除了一般的初始化文件之外，每个用户标识还拥有自己个人的能够定制的初始化文件。第一次登录时，您的初始化文件中包含的是系统管理员设置的默认命令。随着经验日益丰富，您就可以修改这些文件来满足您的爱好。例如，您可以让 Unix 在您每次登录时执行一个特定的程序。

我们将在第 14 章中讨论这些文件，也就是在讨论了 shell 使用过程中包含的细节和基本概念之后(第 2 章中讲过，shell 是读取和处理命令的程序)。

4.5 着手工作：shell 提示

一旦初始化命令结束执行，您就准备好开始工作了。Unix 将启动 shell，并将控制交给它。然后 shell 显示一个提示——称为 **shell 提示**，并且等待您输入命令。接下来，您要键入命令并按<Return>键。shell 会恰当地处理命令(通常通过运行一个程序)。

例如，如果您输入了启动电子邮件程序的命令，那么 shell 将观察该程序是否正常启动，并将控制传给该程序。当电子邮件程序终止时，控制将返回 shell，然后 shell 显示一个新

提示并等待您输入另一条命令。

最终,当再没有命令需要输入时,您可以通过注销(下面解释)结束工作会话,此时 shell 将停止运行。

知道 shell 正在等待输入命令非常重要。基于这一原因,应该将 shell 提示选择为与众不同的。

在 Unix 世界中,有许多不同类型的 shell,而且实际提示依赖于所使用的 shell。目前 3 种最流行的 shell(以流行程度排列)是 Bash、C-Shell 和 Korn Shell。本书后面将详细讨论每一种 shell(第 12~14 章)。现在,我希望您知道的全部内容就是每种特定 shell 的基本 shell 提示是什么样的。

对于 Bash 和 Korn Shell 来说,提示是一个美元符号:

```
$
```

对于 C-Shell 来说,提示是一个百分比符号:

```
%
```

如果系统管理员已经定制了环境,那么提示可能有点不同。例如,它可能显示您登录的机器名称,如下所示:

```
nipper$
```

在这个例子中,提示向我们显示登录的机器是 **nipper**。随着水平的日益提高,您可以用多种不同的方法定制 shell 提示。但是,永远不能修改的就是提示的最后一个字符(\$或者%)。这个字符是一个提醒,提醒您 shell 正在运行,等待键入命令。而且,因为不同的 shell 使用不同的提示,所以这个字符也是正在运行的 shell 类型的指示符(Bash 和 Korn Shell 虽然都使用\$提示,但是记住您正在使用它们中的哪一个并不困难,因此在实践中,它们不会产生混淆)。

无论您正在使用什么 shell,一旦看到提示,您就可以键入任何希望键入的命令,并按下<Return>键。如果您是第一次登录,并且希望练习一下,那么您可以试一下显示时间和日期的 **date** 命令、显示您的用户标识的 **whoami** 命令或者显示所有当前登录系统的用户标识的 **who** 命令。如果希望监听更多的内容,则可以试一下 **w** 命令。该命令告诉您谁登录了系统以及他们正在做什么。

4.6 注销(停止使用 Unix): logout、exit、login

当您结束使用 Unix 时,需要通过注销(log out)结束会话(当我们以名词或者形容词引用这一思想时,我们会使用一个单独的单词 logout)。注销告诉 Unix 您结束了当前用户标识的工作。一旦注销完成,Unix 将停止 shell 并结束工作会话。

当您结束使用 Unix 系统时永远不要忘记注销,这一点很重要。如果您只是走开,使您的终端(或者计算机)保持登录,那么任何人都可以访问您的账户,并以您的用户标识的名义使用 Unix 系统。

如果这样的话，至少会有这样的风险，即以您的用户标识做一些恶作剧。在极端情况下，一些危险的人可能会删除文件(包括您的)，从而导致各种各样的问题。如果发生了这种情况，您需要承担一些责任：使终端在公共位置保持登录就像一辆没有上锁的汽车将钥匙留在点火开关上一样。

注销的方法有若干种。首先，您可以等待直到看到 shell 提示，然后按<Ctrl-D>组合键(按下<Ctrl>键的同时按<D>键)。

当您按下<Ctrl-D>组合键时，它将发送一个 **eof** 或者“end-of-file”信号。实质上，这告诉 shell 不会再有数据了。然后 shell 终止，Unix 将您注销(我们将在第 7 章中详细讨论 Unix 键盘)。

您稍后还会发现，end-of-file 信号还有其他应用，您完全有可能经常按下<Ctrl-D>组合键，不经意地将自己注销。

基于这一原因，Unix 系统提供了一种防护措施。大多数 shell 定义了一种方式，用来指定不希望通过按组合键<Ctrl-D>来注销系统。相反，您必须输入一个特殊的命令。通过这种方式，您就不可能偶然地注销系统。

或许系统管理员已经将系统设置成这样，所以默认情况下，您不可能通过按组合键<Ctrl-D>来注销系统。如果是这种情况，那么您必须使用特定的注销命令。它们是 **logout** 和 **exit**。

为了查明如何才能注销系统，您可以尝试按下组合键<Ctrl-D>。如果它起作用，则可以通过这种方式注销系统。如果它不起作用，那么您的 shell 已经设置为忽略<Ctrl-D>组合键了，您可能会看见如下所示的消息：

Use "logout" to logout.

在这个例子中，使用的是 **logout** 命令(键入“logout”然后按<Return>键)。如果您看见如下所示的消息：

Use "exit" to logout

那么您需要使用 **exit** 命令。

注销的最后一种方法是使用 **login** 命令。这告诉 Unix 您要注销，并且准备以一个新用户标识登录。在您注销之后，Unix 将显示最初的提示请求键入一个新用户标识：

login:

如果您想注销，同时又希望使计算机或者终端做好其他人登录的准备，那么这条命令相当方便。

提示

在一些系统上，**login** 命令并不能让您完全脱离系统。这时 **login** 将临时改变用户标识，但是您仍然以原始名称处于登录状态。当新用户注销时，他将回到您的原始会话中。

如果您的系统是这种情况，那么就不应该使用 **login** 进行注销，因为它允许其他人在您的用户标识下结束登录。

您可以通过测试 **login** 来确认系统中 **login** 的工作方式。输入 **login** 命令，然后登录再注销，查看您是否回到原始会话中。如果是这样的话，使用 **login** 进行注销就是不安全的，请使用 **logout** 和 **exit**。

4.7 大写字母和小写字母

Unix 区分小写字母和大写字母。例如，在讨论可能的用户标识时，我使用 **harley** 和 **hahn** 作为示例，这两个例子都以小写的“h”打头。同时，我还建议了一个可能的口令 **H!lg%12**，它包含两个小写字母和一个大写字母。

一些操作系统被设计为忽略小写和大写字母之间的区别，一个著名的例子就是微软的 Windows。而 Unix(历史相对较长)要求更高的精确性。

出于方便考虑，我们称小写字母为 **lowercase**，大写字母为 **uppercase**。这两个名称来源于电传打字机术语。当您使用一台旧式的电传打字机时，按<Shift>键将转换到“upper”字母状态，就可以打印大写字母。

提示

大写字母和小写字母的思想只应用于字母表中的字母，不应用于标点符号、数字或者特殊字符。

在 Unix 中，当您键入名称或者命令时，必须确保完全准确。例如，如果您的用户标识是 **harley**，那么在您登录时，必须键入全部小写的字母。如果键入 **Harley**，那么 Unix 将认为它是一个完全不同的用户标识。同理，当您注销时，必须键入 **logout**，而不是 **Logout**。

当程序或者操作系统区别大写字母和小写字母时，我们就称它区分大小写(case sensitive)。因此，我们可以说 Unix 区分大小写，Windows 不区分大小写。

因为 Unix 认为大写字母不同于小写字母(实际上它们也确实不同)，所以系统管理员可以分配两个完全不同的用户标识，而这两个用户标识只在字母的大小写上不同，例如 **harley** 和 **Harley**。但是，在实践中，您可能永远不会见到这样的用户标识，因为这样十分容易引起混乱。实际上，用户标识习惯上只使用小写字母。

请注意大写字母和小写字母之间的区别只适用于登录和 Unix 命令的输入过程。当使用处理普通文本数据的程序时，例如使用字处理程序创建文档时，可以按常规模式键入其内容。

4.8 Unix 会话样本

图 4-3 示范了 Unix 的一个简短会话。这个例子使用一个共享系统创建，该共享系统上同时有多个用户登录。

会话从使用用户标识 **harley** 登录开始，注意 Unix 不回显口令。

在用户标识和口令接受之后，Unix 系统标识自己。在这个例子中，您可以发现我们使用的运行 Linux 的计算机叫 **weedly**。

剩余的数字显示内核的信息。内核的版本是 **2.6.22-14-generic**，它于 6 月 24 日下午 4:53 分构建(记住，Unix 使用 24 小时制时钟)。

```

login: harley
Password:

Linux weedly 2.6.22-14-generic Tue Jun 24 16:53:01 2008

The programs included with the Ubuntu system are free
software; the exact distribution terms for each program
are described in the individual files in
/usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

Last login: Sat Sep 20 08:33:17 from nipper.harley.com

harley@weedly:~$ date
Mon Sep 29 10:34:36 PDT 2008

harley@weedly:~$ who
tammy  tty1    Sep 28 21:25
linda  pts/0    Sep 29 07:13 (static)
tammy  pts/1    Sep 29 09:31 (coco)
casey  pts/2    Sep 29 10:07 (luna)
alex   pts/4    Sep 29 10:27 (alpha.taylored-software.com)
harley pts/3    Sep 29 10:34 (nipper.harley.com)

harley@weedly:~$ logout
Connection to weedly.wordsofwonder.net closed.

```

图 4-3 Unix 工作会话示例

接下来的两条消息与 Ubuntu 有关，Ubuntu 是一种特殊 Linux 发行版的名称。

在 Ubuntu 消息之后是一行显示该用户标识上一次登录的时间的信息。上一次登录时间是 9 月 20 日上午 8:33 分，而且连接来自于一个命名为 **nipper.harley.com** 的计算机。

最后，说明信息结束，显示一个 shell 提示。在这个例子中，提示被配置成显示用户标识的名称(**harley**)、计算机的名称(**weedly**)和 **\$** 字符。

\$ 表示使用的 shell 是 Bash，而且 shell 已经做好输入命令的准备。

我们输入 **date** 命令，该命令将显示当前的时间和日期(Unix 中确实有一个 **time** 命令，但是它并不显示时间，而是记录特定命令执行的时间)。

在 **date** 命令显示输出之后，我们再次看到 shell 提示。然后，我们输入 **who** 命令。这个命令显示当前登录到系统的所有用户标识的列表。

第一列显示用户标识。注意 **tammy**(正好是系统管理员)从两个终端登录。

第二列内容 **tty1**、**pts/0**、**pts/1** 等是所使用终端的名称。

第三列显示该用户标识的登录时间。

最后一列显示用户登录时使用的计算机。前 3 台计算机(**static**、**coco**、**luna**)和 **weedly** 位于同一个网络上。最后两台计算机位于远程网络上，所以以较长的名称显示。

在 **who** 命令结束之后，可以看到另一个 shell 提示。我们键入 **logout** 命令，结束会话。

正如前面所述，用户标识 **tammy** 登录了两次。Unix 允许您不用注销就登录任意多次。但是，在某一时刻您通常只能使用一个终端。在我们的例子中，**tammy** 是系统管理员，所以没有问题。

但是，如果您输入了 **who** 命令，发现自己从不止一个终端登录，那么您应该断定发生了什么事情。您可能不经意间结束了一个以前的工作会话而没有注销。或者您可能运行了

不止一个 X 会话(参见第 6 章), 而且每个这样的会话都以一个单独的终端显示。另外还有一个解释, 但是可能没有那么亲切: 可能有人在没有获得您的授权的情况下使用您的用户标识登录系统。

4.9 改变口令: `passwd`

在建立 Unix 账户时, 系统管理员将为您分配一个用户标识和口令。系统管理员通常按他们自己的方式组织事情, 因此您有可能无法获得自己希望的用户标识。

例如, 您可能希望使用自己的名字作为用户标识, 但是系统管理员决定所有的用户标识都使用姓氏。千万不要与系统管理员发生冲突, 系统管理员拥有大量的责任——Unix 系统很难管理——而且还有可能劳累过度。

您可以改变自己的口令。实际上, 出于安全考虑, 一些系统管理员会使用一个称为口令过期时间(password aging)的功能强制您定期改变口令(口令过期时间功能还可以用来防止您过于频繁地改变口令)。例如, 系统可能要求您每 60 天改变一次口令。如果您的系统上安装有口令过期时间功能, 而且您的口令已经过期, 那么当您登录时, 系统就会提醒您改变口令, 然后强制您选择一个新口令。

除此之外, 您还可以在您希望的时间自由地改变口令(只要系统管理员没有限制)。在改变口令时, 需要使用命令 `passwd`。

一旦输入了命令 `passwd`, `passwd` 就会要求您键入旧口令。这样可以确认您有权改变口令。否则, 任何一个在已经登录的终端或者计算机旁边的人都可以改变您的口令。

接下来, `passwd` 要求键入新的口令。一些系统要求所有的口令都要满足特定的规范。例如, 口令至少需要有 8 个字符。如果新口令不满足本机标准, 那么系统将通知您并要求输入一个新的口令。

最后, `passwd` 要求您再次键入新的口令。输入新口令两次可以确保在输入口令的过程中没有犯错误。

在键入口令时, 字符是不回显的。这可以防止他人偷看到新口令。

名称含义

`passwd`

为了改变密码, 需要使用 `passwd` 命令。很明显, `passwd` 是“password”的缩写, 那么为什么不将这个命令命名为 `password` 呢?

答案就是 Unix 人喜欢短名称。在学习 Unix 时, 您将碰到许多这种传统。例如, 列举文件的命令是 `ls`, 复制文件的命令是 `cp`, 显示进程(正在运行的程序)状态的命令是 `ps`。这类命令数不胜数。

起先, 省略几个字母看上去并没有必要, 而且还有点古怪, 但是一旦适应了这种方式, 您将会发现这种简洁让人特别舒适。

4.10 口令选择

使用口令的原因就是为了确保只有授权的人才能够访问 Unix 账户。大家可以想象，总是有那么一些聪明的人一直在不断地尝试着潜入系统。这样的人称为骇客(cracker)(通常将麻烦制造者称为黑客(hacker)。黑客和骇客之间有一定的区别，稍后对此进行解释)。

一些骇客只是希望挑战 Unix 的安全系统，看看他们能不能秘密地登录到系统中。还有一些骇客喜欢制造实质性的破坏。

因此，您需要(1)永远不把自己的口令告诉他人；(2)选择一个不容易被猜测的口令。记住，如果您将口令告诉了他人，而他破坏了计算机系统，那么您就要负责任。

当您第一次得到 Unix 账户时，系统管理员已经为您选择了一个口令。无论何时，当您希望时，就可以使用 **passwd** 命令来改变您的口令。

口令选择的规则实际上是不要选取某些种类的口令的指南：

- 不要选择用户标识(例如 **harley**)，或者将用户标识反过来拼写(**yelrah**)。这样就好比将房子的钥匙藏在席子下面一样。
- 不要选择自己的名字或者姓氏，或者两者的组合。
- 不要选择爱人或者朋友的名字。
- 不要选择字典中的单词。
- 不要选择一串对自己有特殊含义的数字，例如电话号码、重要日期(例如生日)、社会保障号等。
- 不要选择与 Harry Potter、Star Wars、Monty Python、The Hitchhiker's Guide to the Galaxy 或者其他与流行文化相关的口令，即便只有一丝关联。
- 不要选择一个键盘序列，例如 **123456**、**qwerty** 或者 **1q2w3e4r**。有些密码猜测程序专门猜测这种类型的口令。
- 不要使用口令 **fred**。许多人选择使用这一口令，因为它容易键入，但它也是黑客首先尝试的口令。

另外，还有几种日常防范措施：

- 永远不将口令写在一张纸上(在您弄丢了之后肯定会有人发现它的)。
- 定期改变口令(一个月一次比较好)。

在骇客社区中，有许多猜测口令的程序。这样的程序不仅可以进行智能的猜测(例如您的名字、姓氏等)，而且还使用一长列可能的口令来看看是否能够匹配成功。例如，一大列的字典单词、名字和姓氏、电影明星、电影名、Internet 地址、美国邮政编码等，更有甚者，有的程序还包括外语单词。

因此，如果您认为某一思想非常出名和有趣，那么黑客有可能在您之前就已经想到。当口令与流行电影、书籍、电视剧相关时，这一结论尤其真实。例如，如果您是一名大学生，Star Wars 和 Monty Python 在您出生之前就已经流行很久了，那么您所能想象的与它们相关的名称或者术语，没有一个不位于早已广泛分发的骇客猜测列表中。

口令破解程序要比您想象的更成功，因此您需要明智地选择口令来保护自己(以及自己的文件)。最好的方法是生成一系列没有意义的字符。为了保险起见，需要混合大写字母、

小写字母、数字以及标点符号(一些系统强制采用这种方式)。作为一个示例,大家可以考虑一下口令 **H!lg%12**, 在本章前面已经使用过这个口令。这样一个口令猜测起来就比较困难。

如果怀疑有人知道您的口令,那么就立即改变您的口令。如果使用的是共享系统,而您忘记了自己的口令,那么您只需告诉系统管理员。系统管理员可以在不知道旧口令的情况下,为您指定一个新口令。然后您就可以根据自己的需要再改变新口令。

一个理想的口令是您不用写下来就可以记住,但是其他人不易猜测,而且还永远不会出现在骇客的单词列表中。一个好的办法就是以对一个对您有意义的短语或者句子作为对象,生成一个缩写。下面举一些例子:

```
dontBL8 ("Don't be late")
2b||~2b (for C programmers: " To be or not to be")
wan24NIK8? (random meaningless phrase)
```

您已经领会了吧?但是您还要确保,在创建了一个很酷的口令之后,兴奋之余,不要将这个口令告诉他人以炫耀您的聪明。

名称含义

黑客、骇客

花费大量时间编程的人有两种类型:黑客和骇客。黑客指那些花费时间在有用(或者至少无害)的编程项目上的人。

单词 **hack** 经常用作动词,表示对编程有传奇色彩的贡献,例如“Elmo spent all weekend hacking at his file-sharing program(Elmo 将全部周末时间都花在他的文件共享程序上)”。

因此,术语“黑客”通常有正面的含义,描述那些像书呆子一样不停钻研的人。此外,“黑客”也指那些知道如何使用计算机创造性地解决问题的聪明人。

尽管不怎么酷,但是他们是对社会有用的人。全世界在经济上最成功的黑客就是比尔·盖茨。

骇客指坏家伙:以破坏计算机系统以及做专家不希望他做的事情为乐趣的人(注意这里我用了“他(him)”。这是因为——或许是遗传因素——几乎所有的骇客都是男性)。

骇客是那些您不希望将妹妹嫁给他的人,而与黑客成为一家人则没什么问题,无非是每个人都通过电子邮件接收结婚邀请。此外在蜜月期,您每天都会接收到一封这对幸福夫妇在做什么的电子邮件,并且还附带一个网址,在这个网址上您可以找到他们旅行的最新照片以及已更新的博客。

4.11 检查他人是否使用过您的 Unix 账户: last

每次登录时,您都要小心地查看开始的消息,大多数系统将告诉您上一次登录的时间和日期。如果您记着没有在那个时间登录过,那么可能有人使用了您的账户。

为了进一步检查,可以使用 **last** 命令。只需输入 **last** 命令并在之后跟着用户标识即可。

例如，如果您以 **harley** 登录，则输入：

```
last harley
```

您将会看到一些信息，告诉您上一次或者最近几次的登录时间。如果只输入了命令 **last** 而没有输入用户标识，即：

```
last
```

那么您将看到系统上所有用户标识的信息。这些信息的生成可能要持续一些时间，因此如果您希望提前结束这条命令，可以按<Ctrl-C>组合键(在按下<Ctrl>键的同时按下<C>键)。

您或许认为在输入没有用户标识的 **last** 命令时非常有趣，因为您可以偷看其他所有人是什么时候登录的。好的，如果您希望，那么您可以这样做，但是我确信很快您就会感觉厌烦。如果真的没事可做，那么使用第 8 章中将要描述的程序或许会使您觉得更加有趣。

4.12 用户标识和用户

用户(user)指以某些方式利用 Unix 系统的人。但是，Unix 本身并不了解用户，它只知道用户标识。

两者之间的区别是一个重点。例如，如果有人使用您的用户标识登录，那么 Unix 不知道是不是真的是您登录了(这就是为什么需要保护口令的原因)。

在 Unix 世界中，只有用户标识拥有真实的身份。用户标识——而不是用户，拥有文件、运行程序、发送电子邮件、登录和注销系统。这意味着如果有人能够使用您的账户登录，那么他将拥有和您相同的权利。他能够修改您的文件，以您的名字发送电子邮件，等等。

在本章前面，我们示范了一个会话，在这个例子中，我们使用命令 **who** 查看谁登录了系统。图 4-4 展示了这个命令的输出。

```
$ who
tammy    tty1    Jun 28 21:25
tlc      pts/0   Jun 29 07:13 (static)
tammy    pts/1   Jun 29 09:31 (coco)
casey    pts/2   Jun 29 10:07 (luna)
harley    pts/3   Jun 29 10:52 (nipper.harley.com)
alex     pts/4   Jun 29 14:39 (thing.taylorred-soft.com)
```

图 4-4 who 命令的输出

注意您只能看到用户标识，而不是人的名字。这是因为 Unix 系统根植于用户标识，而不是用户。

4.13 超级用户的用户标识：root

在 Unix 中，所有的用户标识都或多或少是平等的，只有一个例外。

有时候,系统管理员有必要拥有特殊的权限。例如,他需要在系统中添加一个新的用户、改变他人的口令、升级一些软件等。

按照这个要求,Unix 支持一个特殊的用户标识,称为 **root**,它拥有非凡的权限。使用 **root** 用户标识登录的人可以做任何希望做的事情(很明显,**root** 的口令是一个要高度保守的秘密)。当有人以 **root** 登录时,我们就称他为**超级用户**。

开始,名称 **root** 没有什么特殊含义。但是,在第 23 章中,您就会发现整个 Unix 文件系统的基础称为“**root** 目录”。因此,名称 **root** 指的是 Unix 一个非常重要的部分。

大多数时间,系统管理员使用他自己的正常用户标识进行常规的工作,只有在进行需要特殊权限的工作时才切换到超级用户。一旦特殊任务完成,系统管理员又会切换回他自己的正常用户标识。这可以防止超级用户的强大权力不经意间对系统造成严重损害。

例如,如果您不小心输入了 **rm**(移除)命令,它可能会意外地删除文件。如果以自己的用户标识登录,那么最坏的情况就是删除自己的文件。如果以 **root** 登录,那么一个形式不正确的 **rm** 命令可能会删除系统中任意位置上的文件,造成大范围的损害。

提示

当 shell 做好接受命令的准备时,它会显示一个提示。提示的最后一个字符显示正在使用哪一种 shell。例如,Korn shell 和 Bash 都使用字符\$,而 C-Shell 使用字符%。

不管使用哪一种 shell,当以超级用户登录时,提示都将改变成字符#。当看到#提示时,一定要小心——超级用户拥有极大的权利。

4.14 安全计算实践中体验快乐

在早期时代,Unix 是为那些需要共享程序和文档而在一起工作的人设计的,他们喜欢彼此帮助。系统的基本设计假定每个人都是诚实的,没有恶意的。即使是现代的 Unix,尽管采取了口令和安全措施,也不是百分之百的安全,更不用说过去的 Unix 了。所以一开始必须假定使用 Unix 的人尊重其他用户。

因为 Unix 如此复杂,所以总是有少数几个骇客不停地尝试击败该系统。在一些环境中,允许年轻的程序员尝试着潜入系统并执行一些秘密的动作,他们的天才甚至还会受到赞赏。

在 Unix 社区中不是这样的,骇客和麻烦制造者都会受到追捕和惩罚。前面提到有一些现有的程序可以用来猜测他人的口令,在一些学校里,被现场抓住运行这样的程序的人将被立即驱逐。

但是,Unix 的神奇之处就是 Unix 世界中充满着挑战和乐趣。作为我的读者之一,您不见得会厌烦到非做坏事不可的地步。无论它对您的诱惑有多大,都请记住,系统管理员的工作是很繁重的,他们对故意制造麻烦的人没有耐心。

如果您喜欢 Unix,并且有富余的时间,那么您可以在传授和帮助(两个最重要的 Unix 传统)其他人的过程中获得极大的乐趣。

4.15 练习

1. 复习题

1. 用户和用户标识之间的区别是什么？
2. 注销有哪 4 种不同的方式？
3. 您怀疑有人在没经过授权的情况下使用了您的 Unix 账户。您如何检查这种情况？假定您发现有人使用了您的账户，但是不知道他是谁，您如何停止他们？
4. 超级用户的用户标识是什么？

2. 应用题

1. 改变口令是一项基本的技能，在开始使用 Unix 之前必须掌握。每次改变口令时，都应该立即测试口令以确保口令正常。使用 **passwd** 将您的口令改变为 **dontBL8**(即“Don't be late”)。改变完口令之后注销系统，接着再登录系统对口令进行验证。然后再将口令改变回原来的口令，并再次对口令进行测试，以确定口令正确。

3. 思考题

1. Unix 是区分大小写的，也就是说，Unix 区分小写字母和大写字母。微软的 Windows 是不区分大小写的。例如，在 Unix 中，**harley** 和 **Harley** 是两个完全不同的名称。而在 Windows 中，它们是相同的。您认为最初的 Unix 开发人员为什么要选择使 Unix 区分大小写呢？为什么微软公司选择使 Windows 不区分大小写呢？您更喜欢哪一种方式？原因是什么？
2. 当使用 **passwd** 命令改变口令时，程序要求键入新口令两次。这是为什么呢？
 3. 为什么拥有一个超级用户十分重要？



GUI: 图形用户界面

与 Unix 进行交互的方式有两种：使用基于文本的界面或者使用图形界面。第 4 章中通过解释基于文本界面的共享系统，介绍了 Unix 的使用。在本章中，我准备解释图形界面，具体包括：什么是图形界面、如何以及为何开发它们和现在最常使用的图形界面有哪些。在第 6 章中，我们将讨论两种类型的界面，而且还将介绍如何管理工作会话的细节。

在讨论之前，我希望先介绍一些图形界面的基本概念，即如何看待图形界面以及它们在 Unix 世界中的地位。之后，我将提供几个话题：几个大家可能会听懂的笑话；一个大家可能听不懂的笑话；一个确定您适合使用的图形界面是 KDE 还是 Gnome 的真/假测试(您很快就会明白)；以及一些关于如何创建祖母机器(Grandmother Machine)的合理建议。

5.1 什么是 GUI

图形用户界面或者 GUI 是一个允许您使用键盘、指点设备(鼠标、跟踪球或者触摸板)及显示器与计算机进行交互的程序。输入来自于键盘和指点设备，输出显示在显示器上。界面的设计不仅包含字符，还包含窗口、图形和图标(小图形)，而且所有这些东西都是可操控的。

在显示信息时，广义地讲，有两种类型的数据，即文本(字符)和图形(图像)，因此将其命名为图形用户界面。Microsoft Windows 和 Macintosh 都使用 GUI，因此我确信您对 GUI 已经熟习。

提示

当谈论 GUI 时，“GUI”的发音有两种方式：或者是 3 个单独字母的发音“G-U-I”，或者作为一个单词发音“gooey”。

您可以选择一种适合您的性格和您的听众的发音(我个人将它读作“G-U-I”)。

因为文化的惯性，今天大多数 GUI 都遵循相同的基本设计。与 Windows 和 Mac 相比，当您查看不同 Unix 的 GUI 时，会发现一些重要的区别。Unix 世界中最基本的一点就是没有人相信一种尺寸就能够适合全部对象。作为一名 Unix 用户，您拥有许多选择。

为了使用 GUI, 您需要理解几个基本的思想, 并掌握几项技能。首先, 您需要学习协调地使用两种输入设备: 键盘和指点设备。

大多数人使用鼠标, 但是正如前面所述, 您可能也见过跟踪球、触摸板等。在本书中, 假定大家使用的是鼠标, 但是鼠标和其他指点设备之间的区别很小(顺便说一下, 我喜欢使用跟踪球)。

一般情况下, 随着您移动鼠标, 屏幕上的指针就跟着运动。这个指针是一个小图形, 通常是一个箭头。在一些 GUI 中, 当指针从屏幕的一个区域移动到另一个区域时, 指针形状将发生改变。

指点设备不仅用来移动屏幕上的指针, 而且还有可以按的按键。Microsoft Windows 要求鼠标有两个按键, Mac 只要求一个单独的按键。Unix 的 GUI 更复杂一些。大多数 Unix 的 GUI 基于一个叫做 X Window 的系统(稍后详细解释)。X Window 一般使用 3 个鼠标按键, 尽管其操作也可以使用 2 个鼠标按键完成。

按照约定, 鼠标的 3 个按键按从左向右的顺序编号。按键 1 位于左边, 按键 2 位于中间, 按键 3 位于右边。GUI 设计为按键 1(左边的按键)的使用最为频繁。这是因为, 如果您习惯使用右手, 并且鼠标在您的右手, 那么左边的按键是最容易按的(使用右手食指)。如果您习惯使用左手, 则可以改变按键的顺序, 并将鼠标移动到左边, 用左手使用鼠标。

对于 GUI 来说, 屏幕被分成许多有边界的区域, 这些区域称为窗口。和真实的窗口一样, GUI 窗口的边界通常是矩形的, 但也并不总是如此。与真实的窗口不同, GUI 窗口可以在屏幕上重叠, 而且无论何时都可以改变它们的大小和位置(本章后面的图 5-3 和图 5-4 中将进行示范)。

每个窗口为不同的活动包含输出并接受输入。例如, 您可能使用 5 个不同的窗口, 每个窗口都包含一个不同的程序。在您工作时, 可以方便地从一个窗口切换到另一个窗口, 从而允许从一个程序切换到另一个程序。如果您不希望看到某一个窗口, 可以收缩这个窗口并隐藏它。当您结束使用这个窗口时, 还可以将这个窗口永久地关闭。

在第 4 章中, 我们讨论了使用基于文本界面(一种仿真字符终端的界面)的 Unix 是什么样子的。在这种情况下, 某个时刻只能看到一个程序。在 GUI 中, 一次可以看到多个程序, 并且还可以方便地从一个程序切换到另一个程序。实际上, X Window(概括地讲, 窗口系统)开发背后的主要动机之一就是使人们同时使用多个程序时尽可能地方便。

为了使用 Unix 的 GUI, 还有其他一些重要的思想和技能需要理解, 我们将在第 6 章中讨论。在本章中, 我们讨论与此类系统相关的最重要的思想。我们首先从形成几乎所有 Unix 系统的 GUI 基础的软件开始, 即 X Window。

5.2 X Window

X Window 是一个为使用图形数据的程序提供服务的系统。在 Unix 世界中, X Window 在 3 个方面非常重要。首先, 它是几乎所有 GUI 的基础。其次, X Window 允许在远程计算机上运行程序, 并在自己的计算机上显示完整的图形输出(参见第 6 章)。第三, X Window 使得使用各式各样的硬件成为可能。此外, 还可以同时使用不止一台显示器。

设想一下,您在使用计算机工作,且打开了5个窗口。其中3个窗口运行您自己计算机上的程序,另2个窗口运行远程计算机上的程序。但是,所有的程序都显示GUI提供的图形元素:图标、滚动条、指针、图像等。正是X Window使所有这些成为可能。X Window通过后台工作提供支持结构,从而使需要显示图形数据并接受鼠标或者键盘输入的程序不必考虑细节。

出于方便考虑,我们通常称X Window为X。因此,您可以问一个朋友:“您知道最常用的Unix GUI都基于X吗?”(我知道X是一个奇怪的名称,但是如果您正在和同道中人交往的话,那么您会很快适应该术语。)

X的根源可以追溯回20世纪80年代中期的MIT(Massachusetts Institute of Technology,麻省理工学院)。那时,麻省理工学院为了教学目的,希望构建一个包含图形工作站(功能强大的单用户计算机)的网络。然而,他们所面对的是来自众多不同厂商的不兼容的设备和软件所造成的混乱。

1984年,麻省理工学院制定了Athena计划,这是MIT、IBM(International Business Machines Corporation,国际商用机器公司)和DEC(Digital Equipment Corporation,数字设备公司)研究人员之间的协作计划。他们的目标就是创建第一个标准化、网络化并且与具体硬件独立的图形操作环境。然后这个环境可以用来构建一个大型的、校园级的网络,称为Athena。

为了建立Athena,麻省理工学院需要将大量的异构计算硬件连接到一个已运行的网络中,而且还要以一种适合于学生的方式来做。这要求Athena计划的开发人员将一群与具体厂商相关的复杂图形界面替换为一个单独的、设计良好的界面——一个他们希望成为工业标准的界面。

因为这是一个雄心勃勃的项目,所以他们决定将这个计划以及网络本身用希腊智慧之神Athena命名(Athena是智慧和战争之神,1984年Athena成为一个重要的联盟,一个希望将不同厂商生产的计算设备连接起来的联盟)。

最终,Athena计划在两个重要方面取得了成功。首先,Athena计划的程序员创建了一个与具体厂商无关且适合于网络的图形界面,他们称之为X Window。X Window逐步被广泛接受,后来还成为工业标准(尽管不是唯一的工业标准)。第二,程序员能够构建Athena网络并且成功地部署它,为麻省理工学院社区中数百台计算机提供服务。

X Window的第一个版本(称为X1)于1984年6月发行。第二个版本(X6)于1985年1月发行,并且于第二年的9月出现了第三个版本(X9)(我确信该编号方案对且只对某些人有意义)。X Window的第一个流行版本是X10,于1985年底发行。

至此,X已经开始吸引麻省理工学院之外的注意力。1986年2月,Athena计划向全世界发行了X。这一版本的X称为X103,也就是X Window版本10.3。下一个重要的发行版是X11,于1987年9月面世。

为什么要告诉您这些呢?这是为了描述一个有趣的现象。当一个复杂的软件产品刚开始的时候,它还没有吸引众多的用户。这意味着开发人员不必考虑对许多人造成不便或者“破坏”使用该产品的程序,因此可以快速地对产品进行修改。一旦产品获得了大量的安装,而且程序员已经编写了大量依赖于该产品的软件,那么对软件进行重大修改就变得异常困难。

产品变得越流行,产品的发展就越慢。当越来越多的人以及越来越多的程序依赖于这

个软件时，对它进行大修改就没有那么方便了。

因此，通过 X Window 的第一个 5 年的发展历程可以看出，X Window 经过了 5 个主要的版本(X1、X6、X9、X10 和 X11)。X10 是第一个流行起来的版本，X11 进一步受到用户的欢迎。X11 是如此成功，导致 X 的开发也极大地减缓了。实际上，20 年过去了，目前 X 的版本仍然是 X11！

确切地说，X11 已经进行了重新修订。毕竟，经过 20 多年的发展，硬件标准已经发生了变化，操作系统也革新了。这些修订称为 X11R2(X Window version 11 release 2)、X11R3、X11R4、X11R5、X11R6，直至 X11R7。X11R7 于 2005 年 12 月 21 日发布(顺便说一下，这一天正好是我的生日)。但是，没有一次修订有足够重要的影响能够使之称为 X12。

自 2005 年起，X11R7 一直是标准。当然，修订一直不断，不过都是相对较小的修订：X11R7.0、X11R7.1、X11R7.2、X11R7.3 等。现在已经有人在讨论 X12 了，但是不会那么快就能完成。

要搞清所有这些现象，可以参考前面提及的一个原则：大量的用户妨碍未来的发展。这千真万确，而且值得牢记，因为这是软件设计的一个重要的长远原则，可是大多数人(大多数公司)没有领会掌握。

但是，还可以从另外一个角度观察 X Window 的发展。您可以说麻省理工学院的程序员设计了这样一个出色的系统，以至于近 20 年之后，其基本原则仍然适用，而且不得不进行的修改主要是在细节方面：修复 bug、支持新硬件以及适应新版本的 Unix。

X 程序员所展现的是，当您开发一个重要的产品而且希望它能持续很长一段时间时，在开头花大量的时间是值得的。一个灵活的、深思熟虑的设计可以使产品的生命力持久。这也是一个编程原则，但是许多人没有领会到这一点。

名称含义

X Window、X

X Window 起源于由斯坦福大学开发的一个特定操作系统。这个系统称为 V，由斯坦福大学的分布式系统小组(Distributed System Group)在 1981 年至 1988 年期间开发。

当为 V 系统开发窗口界面时，这个窗口界面称为 W。一段时间过后，W 程序被送给一名麻省理工学院的程序员。这名程序员以 W 为基础开发了一种新的窗口系统，即 X。

从那时起，程序的名称就没有改变过，原因或许有两方面。第一，Unix 系统的名称通常以“x”或者“ix”结尾，而且 X Window 主要在 Unix 中使用。第二，如果不停地变换名称，那么再有两个字母就到达字母表的末尾了。

最后还请注意，该系统的正确名称是 X Window，不是 X Windows。

5.3 谁负责 X Window

到 1987 年，X 已经非常流行，麻省理工学院希望放弃 X Window 系统维护的责任(毕竟，麻省理工学院是一所学校，而不是一家公司)。起初，一些希望 X 开发保持中立的厂商劝说麻省理工学院继续负责 X Window 的维护。但是，麻省理工学院立场坚定，X 的维

护管理责任最终被传递出去了: 首先是一个组织(MIT X 协会), 接着是另一个组织(X 协会), 然后又传递给另一个组织(Open Group)。

现在, X 由第四个组织维护, 一个称为 X.Org 的独立小组(我打赌您可以猜到它们的网址)。X.Org 成立于 2004 年 1 月, 自 X11R6.5.1 起开始负责 X 的维护。

1992 年, 3 名程序员启动了一个项目, 为 PC 开发一种新版本的 X。具体而言, 该系统运行在使用 Intel 386 处理器的 PC 机上, 因此他们称这一软件为 XFree86(这一名称是个双关语, 因为“XFree86”的发音听起来像“X386”)。

因为 XFree86 支持 PC 视频卡, 所以 Linux 开始使用它。随着 Linux 的日益流行, XFree86 也流行起来。在 20 世纪 90 年代末和 21 世纪 00 年代初期, 当官方的 X 开发缓慢前进时, XFree86 开发人员则一直开足马力。实际上, XFree86 如此流行, 以至于曾经在一段时间, 如果您使用安装了 Unix 和 GUI 的 PC 的话, 那么您十有八九使用的就是 XFree86。

但是, 在 2004 年, XFree86 组织的主席决定对 XFree86 的发行许可证进行改变。他的想法是在发行该软件的任何特定部分时, 强制人们信任 XFree86 开发小组。

听起来这像一个崇高的想法, 而且 XFree86 仍然是开放源代码软件, 这一思想永远不会改变。但是, 新的许可证与标准的 GNU GPL 发行许可证(参见第 2 章)不兼容。这使许多程序员以及大多数 Unix 公司烦恼无比, 因为这样将导致可怕的法律问题。

最终的解决方法就是 X.Org 接过 X 的开发, 并以 XFree86 不受新许可证限制的最新版本为起点。最终结果是, XFree86 项目的大多数志愿者程序员都转向 X.Org。

我提这些内容基于下述两点原因。首先, 有时候您可能会看到名称 XFree86, 我希望您知道它的含义。其次, 我希望您能体会到, 在发布开放源代码软件时, 许可证的细节可能会成为软件未来发展的关键因素。软件越流行, 许可证与先前许可证的协调就越重要。本章后面, 当我们讨论一个叫做 KDE 的系统时, 还会看到这一点。

5.4 抽象层次

前面已经解释过, X Window 是一种可移植的、与硬件无关的窗口系统, 可以运行在许多不同类型的计算设备上。此外, X 可以运行在几乎所有类型的 Unix 以及特定的非 Unix 系统(例如 Open VMS, 最初由 DEC 公司开发)上。

这是如何实现呢? 一个图形窗口系统是如何运行在如此众多的操作系统以及不同类型的计算机、视频卡、显示器、指点设备等之上呢?

正如前面所讨论的, X 作为 Athena 计划的一部分, 其开发目标是提供一个可以在许多不同类型硬件上运行各种软件的图形操作环境。因此, 从一开始, X 就被设计得很灵活。

为了实现这一目标, X 的设计人员使用了计算机程序员所谓的抽象层次(layer of abstraction)。这一思想就是依靠层次定义一个大的整体目标, 层次可以形象化为从底部向上的堆叠, 一个层次堆在另一个层次的上面。每个层次为其上一层次提供服务, 从其下一层次请求服务。层次之间没有其他交互作用。

下面快速地看一个抽象例子, 然后再讨论具体的事情。假设一个计算系统由 5 个层次构成: A、B、C、D 和 E。层次 E 位于底部, 层次 D 位于 E 之上, 层次 C 位于 D 之上,

依此类推。层次 A 位于顶部。

运行在层次 A 中的程序调用运行在层次 B(而且只能是层次 B)中的程序,以执行不同的服务;层次 B 中的程序则调用层次 C(而且只能是层次 C)中的程序;依此类推。

这样一个系统设计好了以后,意味着如果一名程序员工作在层次 C 上,那么他不必了解其他的层次以及它们的功能。他所需知道的就是如何调用层次 D 中的服务,以及如何为层次 B 提供服务。因为他只关心自己所在层次的细节内容,所以他不用在意是否有人修改了层次 A 或者层次 E 的内部。

5.5 窗口管理器

为了巩固抽象层次这一思想,下面考虑一个真实的例子。

从一开始, X Window 就被设计成 GUI 和硬件之间的标准化界面。就其本身而言, X 还不能提供图形界面,也没有任何描述用户界面应该是什么样子的规范。提供实际的 GUI 是另一个程序的任务,这个程序就是窗口管理器(window manager)。

窗口管理器控制着窗口和其他图形元素(按钮、滚动条、图标等)的外观和特征。X 充当窗口管理器和实际硬件之间的桥接器。

例如,假设窗口管理器希望在显示器的屏幕上画一个窗口,它向 X 发送请求以及相关的规格(窗口的形状、窗口的位置、边界的厚度、颜色等)。X 绘制窗口,一旦任务完成就向窗口管理器返回一条消息。

通过这种方式,窗口管理器不必知道任何有关窗口如何绘制的事情。这是 X 的工作。同理, X 也不用知道任何有关实际 GUI 如何创建的事情。这是窗口管理器的工作。

由此您可以看到抽象层次的重要性。某个层次上工作的程序员可以忽略所有其他层次的内部细节。在这个例子中,窗口管理器所在的层次位于 X Window 之上。当窗口管理器需要显示内容时,它就调用 X Window 完成工作。

当 Athena 计划发布 X10(第一个流行的 X 版本)时,他们在其中包含了一个命名为 **xwm**(X Window Manager)且还不完善的窗口管理器。在发布 X10R3 时,他们包含了一个新的窗口管理器 **uwm**。而在发布 X11(第一个非常成功的 X)时,包含的是另一个新窗口管理器 **twm**。

(名称 **uwm** 代表 Ultrix Window Manager, Ultrix 是 DEC 公司的 Unix 系统。后来,该名称被改为 Universal Window Manager。名称 **twm** 代表 Tom's Window Manager,因为它由 Tom LaStrange 编写。后来,该名称变为 Tab Window Manager。)

twm 窗口管理器变得非常流行。实际上,它仍然包含在 X11 中,并且是默认的窗口管理器。它无疑是最有影响力的一个窗口管理器。多年以来, **twm** 已经产生了许多派生产品,程序员们对 **twm** 进行修改后生成了自己的窗口管理器(记住,所有的 X Window 都是开放源代码软件,任何人都可以对任何部分进行修改,以创建自己希望的版本)。

在讨论窗口管理器时我提到了 **xwm**、**uwm** 和 **twm**,这是因为它们在历史上相当重要,因此您应该知道它们的名称。从那之后,还有许多其他为 X 编写的窗口管理器,每个窗口管理器都有自己独特的特征、优点和缺点。但是,在众多新创建的窗口管理器之中,只有两个我希望提及的,即 Metacity 和 **kwm**。

这两个窗口管理器特别重要,但是,在解释其原因之前,我需要讨论下一个抽象层次,一个位于窗口管理器之上的层次:桌面环境。

5.6 桌面环境

正如前面所讨论的,窗口管理器负责提供基本的图形界面。就这一点而论,正是窗口管理器使您能够创建窗口、移动并调整窗口大小、单击图标、控制滚动条等。

但是,现代计算机系统所要求的要比基本 GUI 多得多。您需要一个通盘考虑、前后一致的界面。此外,您还希望界面有吸引力、明智并且灵活(就像现实生活中的人一样)。

GUI 的强大来源于能够提供一个工作环境,在这个环境中,您可以以一种易于理解并且满足您需求的方式操纵各种元素。为此,您的界面需要一个底层逻辑,一个允许您在工作时用来解决问题的逻辑。

在 X 的早期,人们直接与窗口管理器交互。但是,由窗口管理器提供的基本 GUI 只有那么多。它不能帮助您完成与现代计算机使用相关的复杂认知任务。后者是一个更复杂的系统的工作,这个系统就是**桌面环境(desktop environment)**,有时候也叫**桌面管理器(desktop manager)**。

这一名称来源于下述事实,即在您工作时,可以将您的显示器的屏幕想象为一个桌面,您将自己正在工作的对象放在它的上面。这一隐喻是在很久以前选择的,那时图形界面刚刚出来,GUI 设计人员觉得将屏幕看作一个桌面对于没有经过培训的用户来说可能比较直观。就个人而言,我认为这一隐喻可能会使人误解和迷惑。我希望在很久之前就将其抛弃*。

桌面环境允许您回答这样的问题,即:我如何启动一个程序?当我不希望看到一个窗口时,如何移动或者隐藏它?当我记不清文件的位置时,如何查找一个文件?如何将图标从一个位置移动到另一个位置上?如何对文件和程序进行优先级设定,从而使那些最重要的文件和程序能够方便地找到?

下面举一个具体的例子。在窗口管理器中可以处理鼠标的移动以及图标和窗口的显示,桌面环境则允许我们使用鼠标拖动图标,并将图标放在窗口上。在做这个动作时,是桌面环境为拖放思想引入了含义。

提示

永远不要愚蠢地认为计算机界面应该“直观”到对初学者立即有用,复杂的任务需要复杂的工具,而复杂的工具需要时间来掌握。

在一些情况下,设计人员有可能设计出一个非智能的界面,从而使没有经验的人也能够立即使用。但是,第一天容易使用的界面,在您有经验之后将不一定是您希望使用的界面。从长远来看,容易学习的界面比要花费时间学习的强大工具更容易使人灰心。

当使用主要是为了容易学习而设计的工具时,我们最终会觉得系统由计算机控制。当使用设计良好、功能强大但需要花费时间学习的工具时,我们最终会觉得系统由用户控制。

Unix 就是这种情况。

* 当谈论隐喻时我比其他人更加吹毛求疵。考虑一下,美国幽默家 Will Rogers 曾经说过:“I never met-a-phor I didn't like(我永远不隐喻我不喜欢的)。”

5.7 抽象层次：继续

为了继续我们的抽象层次模型，假设在抽象层次中，窗口管理器位于 X Window 之上，桌面环境位于窗口管理器之上。

图 5-1 示范了支持典型 Unix GUI 的抽象层次。注意该图中有几个是我还没有提到过的层次。在底部，X Window 调用操作系统的设备驱动程序与硬件进行实际的通信。在顶部，用户及其运行的程序在需要时调用桌面环境。



图 5-1 抽象层次

在 Unix 中，可以认为图形工作环境是一个包含各种层次的程序和硬件的系统。在最顶部的层次，是应用程序(包括实用工具)以及用户。下一层次是桌面环境。桌面环境下面是窗口管理器，等等。在最底部是实际的计算机。从哲学上讲，我们可以认为整个系统是人类硬件和计算硬件之间的桥接。

这个模型有两个重要的概念我希望强调一下。首先，正如前面所讨论的，在特定层次所发生事情的细节与其他任何层次完全无关。其次，只有相邻层次之间才使用明确定义的接口发生通信。

例如，窗口管理器只与它下面的 X Window 和它上面的桌面环境进行通信。这足以满足要求，因为窗口管理器不须关心其他任何层所发生事情的细节。它只响应上一层(桌面环境)的请求，并且调用下一层次(X Window)来服务它自己的请求。

5.8 Unix 公司如何发展图形界面

当 X Window 刚开发出来时，依今天的标准来看，只有窗口管理器来提供基本的 GUI。随着时间的流逝，人们希望一个功能完全的桌面环境的思想逐渐形成，程序员在界面设计和实现方面也不断地深入。尽管窗口管理器何时被桌面环境取代没有一个准确的时间，下面还是大概地介绍一下它是如何发生的。

到 1990 年时，Unix 世界支离破碎，因为有许多不同的 Unix 公司，每家公司都有自己的 Unix 类型。不管如何承诺协作，公司之间还是存在大量的竞争，大多数公司更加热衷于主宰市场，而不是合作。与年青人在操场上比赛时分队一样，Unix 公司之间也形成了两个庞大的组织，每个组织都声称自己正在开发真正的 Unix。

正如第 2 章中讨论的，在 20 世纪 80 年代中期，大多数类型的 Unix 都或者基于 AT&T

公司的 UNIX, 或者基于伯克利的 BSD, 或者基于两者。在 1987 年 10 月, AT&T 公司和 Sun 公司断然宣布它们准备一起合作, 统一 UNIX 和 BSD。这使其他 Unix 厂商感到不安, 1988 年 5 月, 8 家 Unix 公司为了开发它们自己的“标准” Unix, 组建了开放软件基金会 (Open Software Foundation, OSF)。这 8 家厂商包括 3 家最重要的 Unix 公司: DEC 公司、IBM 公司和惠普 (Hewlett-Packard, HP) 公司。

OSF 的组建刺激了 AT&T 公司和 Sun 公司。它们觉得如果准备与 OSF 竞争, 那么它们也需要自己的组织。因此, 在 1989 年 12 月, 它们控制几家较小的公司一起组建了 Unix 国际 (Unix International, UI)。

因此, 在 20 世纪 90 年代初, Unix 世界有两个竞争的组织, 每个组织都希望自己创建的 Unix 成为真正的 Unix。作为它们工作的一部分, OSF 和 UI 都开发了自己的窗口管理器。OSF 的窗口管理器称为 **mwm** (Motif window manager), UI 的窗口管理器叫 **olwm** (Open Look window manager)。这意味着 X Window 用户在窗口管理器上现在有 3 个流行的选择: **mwm**、**olwm** 和 **twm** (我在前面提到过)。

其中, **twm** 是一个简单的窗口管理器, 而 **mwm** 和 **olwm** 更复杂和强大。实际上, 它们是今天高级桌面环境的祖先。

那么, 为什么 Motif 和 Open Look 没成为今天最重要的 GUI 呢? 答案就在于它们的发起者 OSF 和 UI, 它们在争斗上花费了大量的时间, 从而失去了在 Unix 世界的领导地位。具体详情非常烦人, 因此这里不再详细说明*。最重要的是, 在 20 世纪 90 年代中期, Unix 世界出现了一个大的隔阂, 而这一隔阂被微软的 Windows NT 以及 Linux 填充。随着 Linux 的到来, 又出现了两个新的 GUI, 即 KDE 和 Gnome, 它们与 OSF 和 UI 都没有关系。

5.9 KDE 和 Gnome

1996 年, Tübingen 大学的一名德国学生 Matthias Ettrich 对当前的 Unix GUI 现状感到不满。1996 年 10 月 14 日, 他在 Usenet 上提交了一个贴子, 建议通过启动一个新项目来修补 Unix GUI 的问题, 这一项目叫 Kool Desktop Environment (KDE)。参见图 5-2。

Ettrich 列举各种理由说明当前的窗口管理器还不完善, 他说: “GUI 应该提供一个完整的图形环境。它应该允许用户使用它来完成每天的工作, 例如启动应用程序、阅读邮件、配置桌面、编辑文件、删除文件、查找图片等。所有的部分都必须集成在一起,



图 5-2 Matthias Ettrich, KDE 项目的创始人

Matthias 于 1996 年 10 月创立了 KDE 项目。最终, KDE 取得了巨大的成功, 从而使 Matthias 被视为桌面环境之父。

* 如果真的想知道发生了什么事情, 那么您只需到一个 Unix 编程会议上去, 找一个老一点的人, 邀请他喝一杯。一旦他进入角色, 就请求他告诉您有关“Unix 战争”的故事。

如果无法确定问谁, 那么您可以在会议上来回地寻找, 直到看到一个留着马尾辫、穿着已经褪色的 Grateful Dead T 恤衫的人。

密切地配合工作。”

Ettrich 在为他的女朋友配置 Linux 系统时已经注意到这些不足。他意识到不管自己多么有经验，也没有办法很好地将 GUI 集成到 Linux 中去，从而方便女朋友的使用。他邀请其他人志愿参与 KDE，并许诺“KDE 的主要目标之一就是为所有应用程序提供一个现代的并且拥有标准外观和体验的 GUI。”*

更具体地说，就是 Ettrich 邀请人们帮助创建一个控制面板(带有“漂亮”的图标)、一个文件管理器、一个电子邮件客户端、一个易于使用的文本编辑器、一个终端程序、一个图像查看器、一个超文本帮助系统、多种系统工具、游戏、文档以及许多其他小工具。Ettrich 的邀请受到许多程序员的响应，KDE 项目成立了。

Ettrich 的主要抱怨之一就是流行的 Unix 应用程序在工作或者外观上都不相似。经过 KDE 程序员的勤奋工作，到 1997 年初，他们发布了一些在集成桌面环境中紧密工作的大型的、重要的应用程序。通过这种方式，他们开发的新的、高度功能化的 GUI 开始吸引大众的目光。

几个月后，Linux 社区中的许多程序员开始关注 KDE。Ettrich 选择使用了一个称为 Qt 的编程工具套件来构建新的桌面环境。Qt 由一家挪威人开办的公司 Trolltech 编写，该软件的许可证采用这种方式，即个人可以自由使用，但不能用于商业用途。

对 KDE 程序员来说，这已经足够了，因为他们从一开始就将 KDE 视为一个非商业的产品。但是，其他人觉得 Trolltech 公司的许可证安排还不够“自由”。特别是那些与 GNU 项目和自由软件基金会相关的程序员希望为 KDE 提供一个更少限制的许可证，或者在 GNU GPL 许可证的授权下创建另外一个 KDE(参见第 2 章中有关自由软件的讨论)。

1997 年 8 月，两名程序员 Miguel de Icaza 和 Federico Mena 启动了一个新的项目，来创建另一种 KDE，他们称之为 Gnome。尽管 KDE 早已建立好，但是 Gnome 还是吸引了许多注意力，一年之内，参与 Gnome 的程序员就有大约 200 名。

(您可能还记着，在本章前面提到过两个窗口管理器 Metacity 和 kwm。那时，我说过在众多可用的窗口管理器中，这两个窗口管理器非常重要，我希望大家记住它们的名称。它们之所以重要的原因在于 Metacity 是 Gnome 的窗口管理器，而 kwm 是 KDE 的窗口管理器。)

名称含义

KDE、Gnome

构建 KDE——第一个基于 X 的桌面环境——的项目由一名德国大学生 Matthias Ettrich 启动。那时，Ettrich 提出这一项目，并且建议将该项目命名为 KDE，代表 Kool Desktop Environment。但是，后来 KDE 变成代表 K Desktop Environment。

与 X Window 变成 X 一样，字母 K 也经常用于代表 KDE 桌面环境。例如，在 KDE 中，默认的网络浏览器是 Konqueror，CD 音轨抓取工具是 KAudioCreator，计算器程序称为 KCalc，等等。或许正是因为异乎寻常地使用了字母 K，所以 KDE 的终端仿真器称为 Konsole。

* 推测来说，Ettrich 的女朋友并不像他一样喜爱技术，这使他意识到，尽管程序员可以忍受当前的 GUI，但是它并不适合普通大众。最终，Ettrich 的经历激发了 KDE 项目的产生，从而开发了第一个集成的桌面环境，永远改变了人们对 GUI 的观点。

人们可能想知道：假如 Ettrich 的女朋友没有那么漂亮并且还有点书呆子，那么开发一个真正的桌面环境还需要等多长时间呢？既然 KDE 的到来对 Linux 在全世界的流行有着极其深刻的影响，那么是不是不用争论，应该有更多的社会资源致力于鼓励漂亮的女人与程序员约会呢？

在 Gnome 以及 GNU 项目(总的来说)中, 可以看到字母 G 也存在相同的情况。例如, 与 Photoshop 相似的程序称为 Gimp(GNU image Manipulation Program), 即时通信程序称为 Gaim, 而计算器程序称为 Gcalc, 等等。

名称 Gnome 代表 GNU Network Object Model Environment。“Gnome”或者发音为“Guh-nome”, 或者发音为“Nome”。以我的经验来看, 编程极客在发音 GNU 时是一个硬腭音 G(“Guh-new”), 在发音 Gnome 时是一个软腭音 G(“Nome”)。

5.10 CDE 和总拥有成本

到 1999 年时, KDE 和 Gnome 成为两个流行的、设计良好的桌面环境。在 Linux 社区中, 两个 GUI 都受到广泛的支持(一直到今天, 它们仍然被全世界广泛使用)。

在此期间, 商业 Unix 公司仍然忙于生意, 此时, 它们意识到桌面环境的重要性。前面提到的 Open Group 组织已经接管了 Motif 窗口管理器的开发。在 20 世纪 90 年代初期, 它们开始基于 Motif 开发一种新的专有桌面环境——CDE(Common Desktop Environment)。在经过众多公司的大量努力之后, 1995 年 CDE 面世。到 2000 年时, CDE 成为商业 Unix 系统(例如 IBM 公司的 AIX、惠普公司的 HP/UX、Novell 公司的 Unix 以及 Sun 公司的 Solaris)的 GUI 选择。

您可能会奇怪为什么还需要 CDE 呢? 为什么在 KDE 和 Gnome 都可以免费使用时, 众多计算机公司还要花钱开发专有的产品呢? 为什么许多公司还大规模地需要商业 Unix 呢? 毕竟, Linux 可以自由使用, 并且许可证条款没有约束。难道这些公司不能转向 Linux 并使用 KDE 或者 Gnome 吗?

答案与商业市场和消费者市场的一个基本区别有关, 这一区别非常重要, 因此我希望花点时间来解释它。

作为消费者, 您和我希望软件满足两个条件。首先, 它不能太贵(如果可能的话, 最好免费); 其次, 它能正常工作。当遇到问题时, 我们需要自己去解决。我们可以阅读文档, 可以在 Internet 上查找帮助, 或者请求他人帮助。如果我们感到情况严重的话, 还可以付钱请他人帮助我们, 不过在大多数情况下, 情况都不会那么紧急。如果我们必须等待解决方案, 那么这只是有点不方便, 并不会造成毁灭性后果。

在公司中, 特别是在大型公司中, 情况就有所不同。即便是一个简单的软件问题也有可能影响数百人或者数千人。等待解决方案的代价可能太过昂贵, 对公司和公司的客户都是如此。因为大型公司经不起严重的问题, 所以它们雇佣专职计算机人员维护网络、服务器以及个人计算机。基于这一原因, 当公司评估产品时——无论是软件还是硬件——它们不关注初始费用, 它们关注的是所谓的**总拥有成本**(total cost of ownership)或者 TCO。

为了计算总拥有成本, 公司必须回答下述问题: 如果决定使用该产品, 那么从长远来看它需要什么开销?

TCO 的计算非常复杂并且非常基本, 就像桌面环境一样。对您或者我来说, 初始开销是唯一的开销。如果我们可以免费地获得 KDE 或者 Gnome, 那么这就是我们所关心的全部。软件问题可能有点令人讨厌, 但是, 正如我所说, 这只不过是方便不方便的问题, 而不是金钱的问题。

大型公司的观点有所不同。尽管它们也评估初始购买费用或者许可证费用，但是它们还要执行一个更复杂、更长期的分析。这类计算的详情已经超出了本书的范围，但是理解该思想相当重要，因此本书在此做一个简单的概括。

在公司选定一个重要的硬件或者软件系统之前，它们的财政分析人员会检查所谓的直接开销和间接开销。直接开销包括硬件和软件：初始购买或者租借费用、运行、技术支持以及管理费用。间接开销与生产力损失相关。具体包括员工在学习如何使用系统上花费的时间量，员工由于帮助其他员工而浪费的时间量(这种事情经常发生)，以及系统故障或例行维修所造成的停工时间的开销。

一旦所有这些开销都估算出来，就将它们转换为每年的开支——一种包括设备折旧及设备更新费用的计算。然后将每年的开支集成到公司范围的预算中，而从公司长远的发展来看，公司的预算要与公司的规划相协调。

在大多数情况下，当软件或者硬件的总拥有成本计算出来后，我们所发现的与直觉相反：初始开销显得并不那么重要。从长远来看，最重要的还是未来的开支和间接开销。

因此，当一家公司考虑选购新软件时，它们通常不问购买或者许可该产品需要花多少钱。它们会问：这个软件与已有环境的集成度有多高？它如何适应我们的长远计划？它如何为我们的客户提供最好的服务？未来维护它的开销如何？

一旦这些问题有了答案，就可以看出对于企业应用(*corporate use*)而言，最好的软件通常不是为个人或者教育用途设计的自由软件。商业软件必须拥有适合于商业的功能部件。商业软件中必须有一系列维护良好的编程工具；必须有一个根据商业需求(而不是个人需要)明确定义的、长远的发展规划；最重要的是，还必须有出色的文档和高质量的技术支持。这就是为什么大型商业机构倾向于选择商业软件的原因。这也是为什么在企业世界中，Linux 无法取代 Microsoft Windows(或许永远不会)的原因。

这并不是说大型公司永远不使用免费软件。当这样做有意义时它们就会这样做。例如，IBM 公司不仅提供它们自己版本的 Unix(AIX)，而且也提供 Linux。但是，当诸如 IBM 之类的公司提供开放源代码(“免费”的)软件产品时，它们要花费大量的钱，以支持及增强该产品。例如，IBM 公司已经花费了数百万美元在 Linux 的开发和支持上。对于大型公司来说，事实是，没有东西是免费的。

继续返回到桌面的设计，您可以看出为什么在 20 世纪 90 年代，企业世界拥有自己的桌面环境是如此的重要。尽管 KDE 和 Gnome 都非常出色，但是它们没有商业所需的功能部件和技术支持。

这就是为什么要成立 Open Group 以及它们开发 CDE 的原因，而且这也是为什么在 20 世纪 90 年代末企业用户选择的桌面环境是 CDE，而不是 KDE 或者 Gnome 的原因。

进入 21 世纪，因为自由软件越来越重要，Unix 公司开始提供自己版本的 Linux 以及它们自己专有的 Unix。例如，IBM 公司既提供 AIX 又提供 Linux，Sun 公司既提供 Solaris 又提供 Linux，惠普公司既提供 HP-UX 又提供 Linux，等等。

可以预料，这同样意味着 Unix 公司也提供 KDE 和 Gnome。例如，IBM 公司为它们的 AIX 用户提供 CDE、KDE 和 Gnome，Sun 公司同时提供 CDE 和 Gnome，惠普公司也同时提供 CDE 和 Gnome。

当然，这些版本的 Linux、KDE 和 Gnome 并不与您或者我从网络上免费下载的没有支持的发行版相同。它们是商业级质量的产品，提供商业级质量的技术支持(以很高的代价)。

5.11 桌面环境的选择

当使用 Unix 时,您会有一种强烈的感受,即桌面环境与实际操作系统是相互独立的。Windows 和 Mac OS 并不是这样的,这是因为微软公司和苹果公司努力使人们确信计算机的每个重要程序(包括浏览器、文件管理器和媒体播放器)都属于操作系统的一部分。

其实事实并不是这样的:它们只是以这种方式打包。因为您是一名 Unix 用户,所以您可以分清操作系统和其他软件之间的区别,这使您可以随意地问自己:“我希望在自己的桌面上放些什么呢?”

许多公司和学校在计算机工具上进行标准化。如果您属于其中一家公司或者就读于其中一家学校,那么您将不得不按照他们的方式使用桌面环境。但是,如果在自己的计算机上运行 Linux,或者您所在组织允许您选择,那么您可以自由地决定使用哪个 GUI。

那么,哪个桌面环境最适合您呢?

如果您使用 Linux,会有许多免费的桌面环境,因此您有许多选择机会(为了明白我的含义,只需在 Internet 上搜索“desktop environment”即可)。但是,几乎所有的 Linux 发行版都或者提供 KDE,或者提供 Gnome,或者两者都提供,所以我的建议是,除非您有明确的原因选择其他 GUI,否则您应该使用这两个 GUI 中的任意一个(参见图 5-3 和图 5-4。当您看这些图时,请注意 GUI——包括窗口、图标、工具栏等——的外观和组织,而不是窗口中的内容)。

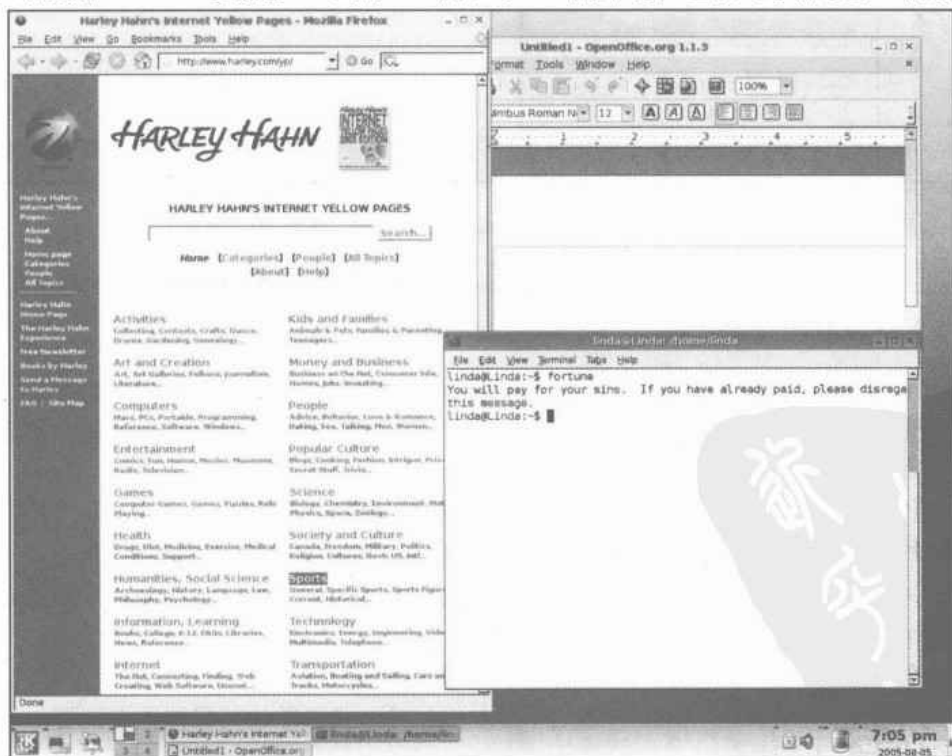


图 5-3 KDE 桌面环境

创建 KDE(第一个真正的桌面环境)的项目由 Matthias Ettrich 在 1996 年启动。他的目标是创建“一个完整的图形环境”,在这个环境中“所有的部分都紧密配合,协调工作”。

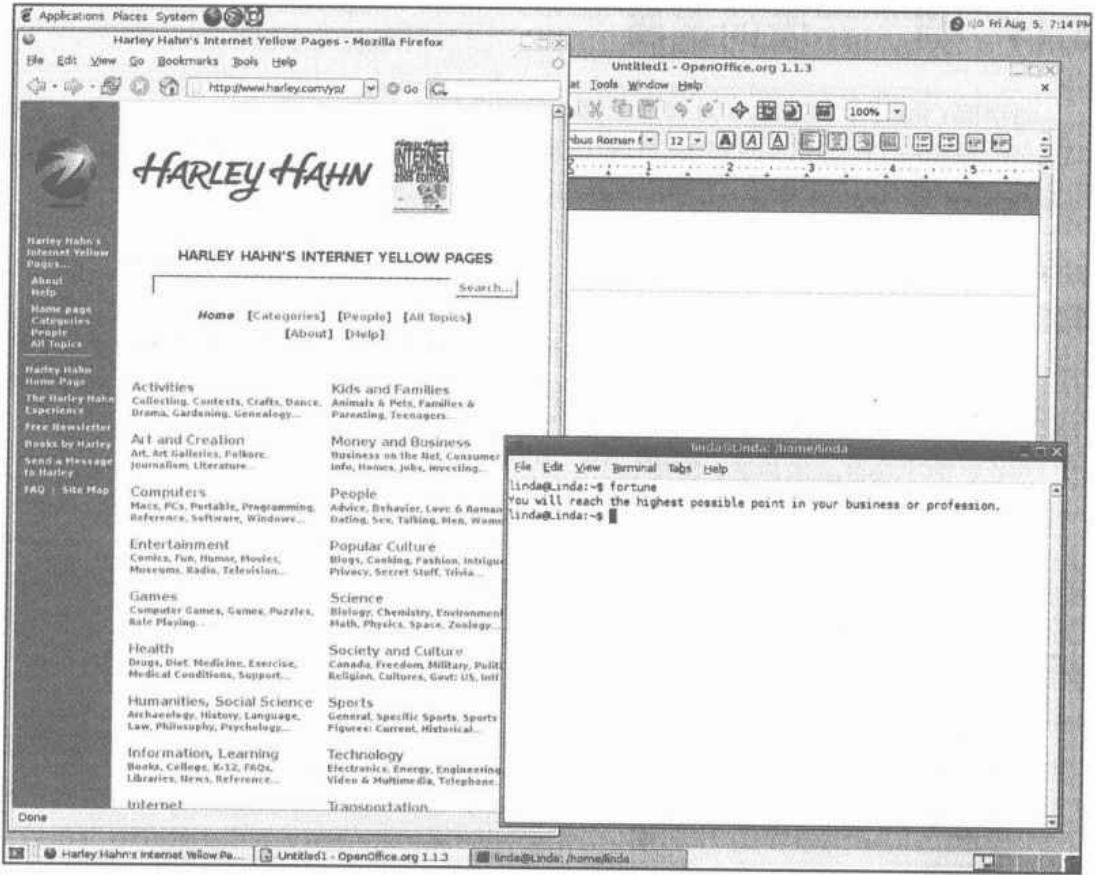


图 5-4 Gnome 桌面环境

Gnome 项目于 1997 年由 Miguel de Icaza 和 Federico Mena 启动，为的是创建另外一种 KDE，以更加自由的许可条款发行。

因此让我们缩小问题的范围：对于 KDE 和 Gnome，哪一个更适合像您这样的人呢？先看看下面的 5 个描述，并根据个人的爱好，将每个描述标记为真或者假。然后我们将评估您的答案，并选择最适合您的桌面环境。

- (1) 我宁愿驾驶手动变速的汽车，也不愿驾驶自动变速的汽车。
- (2) 我在简单而有条理的房间要比装饰豪华但显得混乱的房间感觉更舒服。
- (3) 当我和我的女朋友/男朋友或者妻子/丈夫进行个人讨论时，对我来说重要的是要指出谁对谁错。
- (4) 在购买了 DVD 播放器后，我至少要阅读部分说明书。
- (5) 当使用 Microsoft Windows 或者 Macintosh 时，我通常让其保持原有配置。我不会去乱改颜色、背景等内容。

在解释您的答案之前，我希望您能意识到所有的桌面环境都只是一种使用同一个底层计算环境的方式。因此不管您选择使用哪一种桌面环境，都没有什么问题。但前面已经说过，使用一种适合您个人的桌面环境，您可能会觉得更舒适，因此我们继续分析并选择。

无论您的技术技能或者对计算机的兴趣位于何种层次，如果您回答“真”的问题有3、4或者5个，则选择使用 Gnome；如果您回答“真”的问题只有0、1或者2个，则选择使用 KDE。

注意我已经说过您的选择不应该取决于您对计算机了解多少。一些非常注重技术的人倾向于 KDE，其他高手则倾向于 Gnome。同理，许多没有技术的人选择 KDE，而其他人喜欢 Gnome。

这种二分法与您如何看待世界有关，而不是与您知道多少相关。Gnome 用户倾向于简单和条理。他们希望事情有逻辑。如果需要，他们愿意尽最大的努力使事情以一种对他们有意义的方式运行。

Gnome 用户赞同下述格言：“Form ever follows function(形式永远追随功能)”，这是美国建筑师 Louis Sullivan 于 1896 年提出的一个思想。Sullivan 观察到自然对象的外观受它们功能的影响。Gnome 用户希望世界以一种理性的方式表现，而且他们希望工具能够通过外观直接反映它们的目的。

Gnome 用户喜欢控制事情如何运转，而 KDE 用户喜欢控制事情的外观。这是因为他们对“要正确”的关心要比对他们情感舒适的生活方式的关心更少。

KDE 用户认为这个世界是一个充满色彩、变化，而且有时还充满迷惑的场所。他们倾向于让生活顺其自然，而不是花费大量的时间来修复小的细节。当受到激发投入时间定制自己的工作环境时，他们倾向于使事情更好看，动作更漂亮。

现在，再看看您对上述“真/假”问题的答案。您是 KDE 用户还是 Gnome 用户呢？

这里出现了一个问题。我们有两个不同的桌面环境，每个桌面环境都由一组来自全世界的人一起创建。那么，是否有个人特征来区分 KDE 用户和 Gnome 用户呢？

答案在于每个组的起源。正如前面所讨论的，KDE 小组是由一些不满意当时现状的人启动的。他们希望创建一个完整的工作环境，该工作环境不仅要比那时的窗口管理器好看，而且工作得也更好。

Gnome 小组由一组不满意 KDE 的人启动，因为 KDE 中存在一个与许可条款相关的抽象法律问题，一个大多数人可能会忽略(所有的 KDE 用户都这样)的不足。但是，对于 Gnome 用户来说，或者更加准确地说，对于支持自由软件基金会的人来说，合适的就是合适的，不合适的就是不合适的，所有的都应该是自由的(参见第2章中有关 Richard Stallman 的讨论)。

每个小组的人都创建一种适合自己类型的人的桌面环境显然是有意义的。或许设计消费产品(包括软件)的人都应该更加注意 KDE 和 Gnome 之间的二分法。

5.12 祖母机器

现在我们已经讨论了最重要的 GUI 和最重要的 Linux 类型(第2章)，下面我准备通过回答一个听过多次的问题来结束本章的内容。

“我准备为一个不太懂计算机的人使用自由软件搭建一个系统。我应该使用什么软件呢？”

我称这样一台计算机为祖母机器(Grandmother Machine)，因为它是一种您可能为您外

祖母创建的机器。

当建立起一台祖母机器时，必须意识到您将承担永久的责任，因为无论何时，当您的祖母遇到问题时，她都会叫您。因此，应该使用那些可靠的、容易安装并且容易升级的软件。您还希望能够配置这个系统，从而方便初学者访问网络、查看电子邮件及(在一些情况下)使用字处理软件、电子表格程序、表示图形等。

下面是我的建议。当您阅读它们时，要记住条件随着时间在变化。新软件不断出现，旧软件逐渐变得臃肿，不再适用。因此，要关注我的选项背后的通用原则，而不是具体的选择。

为了创建一台祖母机器，可以使用下述软件。

- **Ubuntu Linux:** 它基于 Debian Linux，且易于安装维护。
- **Gnome:** Gnome 桌面环境使用简单，并且足够健壮，初学者不会把它搞乱。

如果按照前面的问题测试，祖母正好是一个适合 KDE 的人，那么您可以给她安装 KDE。但是，请不要离开 Gnome 或者 KDE。无论您个人喜欢哪个，都不要在不常见的桌面环境上浪费时间。

当安装 GUI 时，一定要花一些时间使祖母能尽可能方便地启动她喜爱的应用程序，最好是在控制面板上创建几个图标(同时，您也可以移除那些她永远不会使用的图标)。

- **Firefox:** Firefox 浏览器易于使用，并且功能强大。至于电子邮件，为她准备一个基于 Web 的账户(例如 Google 公司的 Gmail)，并且让她使用浏览器进行通信即可。

Firefox 是一个神奇的软件，但是一定要花一些时间向您祖母示范如何使用这个软件。另外，还要通过电话回答问题，直到她习惯了使用 Web(避免不必要的问题的最好方法就是创建一个到 Google 的链接，并且向您祖母示范如何使用它。当您这样做时，一定要花一点时间解释如何使搜索结果更有意义)。

- **Open Office:** 一套免费的办公软件(包括字处理程序、电子表格程序等)，与 Microsoft Office 兼容。

最后一点建议：在帮助别人选择计算机系统时我有许多经验，其中有一个普通的原则极少有人提及。

当您为他人选择计算机时，基于“硬件需求”的建议通常并不好用。有效的方法就是基于他们的心理需求来选择系统。该思想非常重要，因此我以提示的形式说明。

提示

Harley Hahn 关于帮助他人选择计算机的原则

(1) 当您为他人选择一台个人使用的计算机时，为他们选择一台满足他们的心理和情感需求的系统。

在大多数情况下，他人可能不会清晰地表明他们的需求，因此您必须自己推断出他们的需求。在这个过程期间，不要陷入硬件清单或者其他琐事的长谈阔论中。

(2) 当您为他人选择一台公司使用的计算机时，选择一个与批准这一开支的人的心理相一致的系统。

在企业环境中，人员流动频繁，因此不要基于特定用户的需求选择系统。选择一台适

合公司的系统即可。

选择系统的最好方法就是挑选一台满足支票填写人的心理和情感需求的计算机，而不是使用这台计算机的那个人。当与小企业打交道时，这一点尤为正确。

5.13 练习

1. 复习题

1. 当谈论信息显示时，广义地讲，有两种类型的数据。这两种类型的数据是什么？
2. 支持大多数 Unix 图形用户界面(GUI)的系统的名称是什么？它最初在何地、何时开发？列举它提供的 3 个重要服务。
3. 什么是抽象层次？列举典型 Unix GUI 环境的 6 个层次。
4. 什么是总拥有成本？谁使用这一概念，为什么？当总拥有成本计算出来之后，初始开销有多重要呢？
5. 什么是桌面环境？在 Linux 世界中，两个最流行的桌面环境是什么？

2. 思考题

1. 通常，当使用 GUI 时，窗口都是矩形的。这是为什么？使用一个圆形的窗口有意义吗？
2. 您为一家使用 PC 的公司工作，PC 机运行 Windows。该公司已经定型了 Microsoft Office 产品(Word、Excel、Powerpoint 等)。一天您在出席一个会议，在这个会议上，一个年轻的、刚毕业的程序员提议公司将 Office 改变成自由软件 Open Office。为什么这是一个坏主意呢？您是否愿意建议公司继续使用微软公司的产品呢？如果愿意，原因是什么呢？



Unix 工作环境

正如第 3 章中解释的，Unix 被开发时使用基于文本的界面。后来，随着更高级硬件的出现，创建了图形界面。尽管 Unix 界面基于硬件，但是它们的创建主要强调理念，而不是它们的外观和基本功能。设计它们是为了提供一个综合的工作环境，这个工作环境要特别适合于人类的心理。

在本章中，将示范如何综合使用文本界面和图形界面。我的目标是向您传授足够多的知识，从而使您可以以一种适合您的思考过程和性格的方式来组织工作。通过这种方式，我准备讨论一些对于掌握 Unix 工作环境十分重要的话题，例如，如何复制和粘贴数据、如何以超级用户的身份工作以及如何关闭与重新启动系统。

在本章中，我们将以第 4 章和第 5 章介绍的思想为基础。因此在继续阅读之前，请确保理解了下述概念。

- 第 4 章：用户标识、口令、登录和注销、shell 提示、大写字母和小写字母、系统管理员、超级用户
- 第 5 章：GUI、桌面环境

在阅读本章内容时，前面学习的许多内容将融合在一块，您也将开始按 Unix 的本来方式使用 Unix。

6.1 同时做不止一件事情：I

在第 3 章中，我解释过 Unix 系统是多任务处理系统，这意味着 Unix 系统可以同时运行不止一个程序。

“多任务处理”是一个技术术语，但是现在它还是一个流行词汇，描述允许人们在同一时间集中于不止一件事情的心智处理能力。例如，人们常说：“Women are better at multitasking than men(女人在多任务处理方面要比男人更出色)”，或者 “It’s okay if I talk on the phone while I drive, because I am good at multitasking(我边打电话边开车是没有问题的，因为我擅于多任务处理)”。

然而事实是计算机和人类实际上都不能在同一时间执行两个相似的任务。当涉及到学习如何使用 Unix 界面时，这种认识有重要的作用，因此在继续学习之前，我希望花点时间

讨论一下多任务处理。我们首先从计算机开始。

在计算机系统内, 所谓的多任务处理实际上是一个非常快速的机器快速地执行多个任务, 看上去就像它们同时被执行一样。

在 Unix 中, 我们不谈论程序的执行, 而是谈论进程(process)的执行。进程就是装载到内存中准备运行的程序, 以及程序的数据与跟踪程序状态所需的信息*。

无论何时, Unix 系统都拥有许多活跃的进程, 每个进程为了运行要请求处理器时间(尽管您可能意识不到这一点, 但是有一些进程正在后台运行)。但是, 一个处理器一次只能执行一个进程。这意味着单处理器的计算机在某一时刻只能接受一个请求。多处理器的计算机能够处理多个请求, 但是即使这样, 也不可能在同一时间服务所有的进程。

为了管理如此之多且相互重叠的处理器请求, Unix 使用了这样一种系统, 即允许每个进程轮流使用处理器一段极短的时间的系统。这一段极短的时间称为时间片(time slice)。典型的时间片通常是 10 毫秒(千分之十秒)。

一旦时间片用完, 当前进程就会挂起, 然后一个特殊的服务(称为调度器)决定接下来执行哪一个进程。因为时间片非常短, 处理器非常快, 而且 Unix 调度器非常巧妙地处理所有的事情, 所以看起来就好像 Unix 同时处理多个事情一样。因此, 多任务处理的假象就这样产生了。**

下面讨论一下人类。在特定的环境中, 我们都可以同时处理多件事情。例如, 当我们吃东西时, 可以同时嗅、品尝和咀嚼。同样, 我们可以边说话边走路, 在唱歌的同时弹奏乐器, 在看电影的同时听声音。但是, 我们不能做的事就是同时思考两件事情。

当然, 我们的心理活动可以来回切换, 如果我们的思考过程足够快的话, 那么看上去就好像所有的活动都在同一时间发生。例如, 假设您同时与四个不同的人进行即时聊天。只要您能在一个合适的时间内应答每一个人, 那么他们就不知道您在同时与他们分别进行交谈。

这听起来有点像 Unix 处理多任务的方式。但是, 它们之间有重要的区别。首先, 相比于计算机, 人类从一个任务转向另一个任务非常慢(下一次进行即时通信时, 您可以试着每 10 毫秒切换一个会话)。

其次, 人类的头脑要比计算机复杂得多***, 而且人类的头脑所执行任务的类型也远比计算机所处理任务的类型复杂。

第三, 当 Unix 从一个进程切换到另一个进程时, 它只需要记录很少的信息。而当您从一个任务改变到另一个任务时, 您需要以一种更加复杂的方式重新适应环境。例如, 考虑一下当您从使用网络浏览器切换到查看电子邮件, 再到接电话, 然后又返回到网络浏览器, 最后转到吃果冻油炸圈饼, 想一想在这一过程中您的头脑中必须发生的事情。尽管您没有自觉地意识到自己的心理过程, 但是它们要比 Unix 在多个进程之间切换复杂得多(另

* 进程这一思想是 Unix 的基础。实际上, 在 Unix 系统中, 每个对象或者用文件表示, 或者用进程表示。简单地讲, 文件存放数据或者允许访问资源, 而进程是正在执行的程序。

进程可以分成更小的单元, 称为线程, 线程是一组运行在进程环境中的指令。因此可以将程序的结构描述为, 在一个进程之内, 不止一个线程在同时运行。

** 在第 26 章中, 将更深入地讨论进程并讨论如何控制它们。现在, 我只告诉您, 如果希望知道当前您的系统上有多少个活跃进程的话, 可以使用 `top` 命令(注: 并不是所有的系统上都提供了这条命令)。

*** 目前已知领域中最复杂的东西就是人类的大脑了。

外,即便是最新版本的 Linux 也不知道如何做一个果冻油炸圈饼)。

计算机和人类之间最大的区别在于人类拥有自由的意愿。任何时刻,我们都可以按自己的意愿思考。这意味着,当我们使用计算机时,我们会对需要完成的事情在如何处理方面形成瞬间的策略,而操作系统必须支持这种策略。因为没有办法事先准确知道在某种特定场合人会怎么想,所以在涉及到提供工作环境时操作系统必须是灵活的。

在合理范围之内,应该允许每个计算机用户根据自己的需求组织自己的工作环境。另外,当需要时,他应该能够改变自己的环境,从这一刻到下一刻——随着他的心理过程变化。另外,当用户更有经验时,应该有更完善的工具供他使用。

例如,初学者(或者头脑迟钝的人)通常在他的计算机上同时只做一件事。他让一个窗口占据整个屏幕,而且他几乎只使用鼠标(不使用键盘)来控制发生的事情。结果,除了最简单的任务之外,他不能执行其他任务。

一名有经验(特别是聪明而有创造性)的用户将使用多个窗口,还尽可能地使用键盘,并且组织好每个时刻的工作,从而可以快速高效地移动。如果您曾经看到过某些熟练掌握自己计算机的人的工作情形,就会明白我的意思。

当人们设计用户界面系统时,他们必须考虑到在思考方面,人类有重大的局限。例如,虽然我们可以记住传统的至理名言,但是却很难记住前面刚刚发生的众多事情*,而且我们无法同时集中在两件事情上。

我们能做的就是当条件变化时,在不到一秒种的时间内重新适应环境。因此,妈妈们能够同时看着一个初学走路的孩子、抱着一个孩子、聊着电话并准备晚饭(这也是为什么人们认为他们可以在聊电话时安全地开车的原因)。

我们有限的记忆力以及能够重新适应环境的能力,意味着一个良好的用户界面必须允许我们在我们希望的任意多个任务之间轮流切换,而且一旦在我们习惯了界面之后,还应该以一种我们觉得舒适的方式来完成任务切换。

另外,用户界面还应该能够支持我们的成长。也就是说不论我们的技能有多丰富,界面仍然能够满足我们的需求。

我的观点就是随 Unix 提供的基于文本的界面和图形界面都是经过精心设计的,并且十分灵活,因此当它们充分结合在一块时,它们能够满足所有这些需求,而且还远不止此。基于这一原因,我希望您在阅读本章时记住两个重要的思想。

首先,我希望您理解您准备阅读的内容不仅仅是事情如何运转的描述。我准备示范的系统是经过反复试验发展起来的,是由目前世界上一些最聪明的人设计的。该系统的目标就是为了克服计算机和使用计算机的人的限制,通过这种方式,让人类认为他和计算机实际上都在执行多任务处理。

其次,我希望您花一些时间来实践和掌握本章中所遇到的界面。当您能够综合利用基于文本的界面和图形界面时,Unix 的真正实力才能得以展现。

我相信,总的来说,Unix 是目前拥有最好用户界面的系统(即使与 Microsoft Windows 和 Macintosh 相比)。在您读完本章内容时,希望您能同意我的观点。

* 您能闭上眼睛回想起 10 秒钟前读过的内容吗?

6.2 GUI 和 CLI

在前面的讨论过程中，不时地提到两种不同类型的 Unix 界面，一种是基于文本的，一种是图形的。现在，是时候揭开它们的正式名称了。

图形界面是 GUI(graphical user interface, 图形用户界面)，我们在第 3 章中已经详细讨论过了。Unix 的 GUI 由 X Window、窗口管理器和桌面环境综合创建。

基于文本的界面通常称为**命令行界面(command line interface, CLI)**。下面介绍使用该名称的具体原因。

第 4 章中讲过，Unix 系统基于文本的基本界面比较简单。**shell**(命令处理器)显示一个提示。您输入一个命令。**shell** 完成执行该命令所需的事情。一旦命令处理完毕，**shell** 就显示另一个提示，您可以输入另一条命令，并这样一直轮流循环。

整个过程只使用文本(纯字符)，而且键入命令的行称为**命令行(command line)**。因此，名称“命令行界面”由此而来。

我希望您知道术语 CLI，因为它是 GUI 的对应物，而且在您阅读时会经常看到这个术语，特别是在 Internet 上。但是，当人们交谈时，他们不说 CLI，而是说“命令行”。例如，您可能在某个网站上读过下述内容：“尽管 Groatcake 软件包是为 GUI 设计的，但是它也有 CLI 版本。”

但是，当人们私下讨论界面时，您极有可能听到这样的话：“我开始使用的是 GUI 版本的 Groatcake 软件包，但是现在我更喜欢使用命令行版本的。”更常见的说法还有：“我喜欢在命令行上使用 Groatcake 软件包。”

换句话说，就是当您看到或者听到“命令行”或者当您看到“CLI”时，它告诉您有人正在键入命令，而不是从菜单中选择选项。

GUI 程序非常重要。例如，几乎所有的人都使用 GUI 版的网络浏览器和办公软件，因此我认为您也将会花大量的时间来使用 GUI。

但是，Unix 的大多数功能在命令行之中，因为它提供了一个快速简单的方法来使用数百个不同的 Unix 命令。实际上，一旦离开本章之后，本书的剩余部分将主要集中在命令行程序的使用上。

提示

作为 Unix 用户，您需要的基本技能是使用命令行输入一条又一条的命令来解决问题。

6.3 使用 GUI 登录和注销

在第 4 章中，讨论了当使用传统的基于文本的 Unix 系统登录时所发生的事情。您看到了提示 **login:**，并且键入您的用户标识。然后，看到提示 **Password:**，并且键入您的口令。系统完成登录过程，启动 **shell**，并提供一个 **shell** 提示。当您结束时，您注销系统。

对于 GUI 来说，登录和注销过程比较复杂。我将使用 Linux 解释这一过程。如果使用

另一种类型的 Unix，那么事情可能会有些不同，但是您应该能够推测出这些不同。

在您打开计算机后，Linux 将启动。当系统准备好登录时，您将看到一个登录屏幕。不同的 Linux 发行版，其登录屏幕外观会有所不同。但是，大多数发行版都会向您展示相同的基本元素。

首先，您将看到一个标签为“Username:”的小方框。这个方框对应于 **login:** 提示。这个方框要求您输入用户标识。

在输入用户标识之前，可以花点时间看看登录屏幕。您将看到若干其他选项：Language、Session、Reboot 和 Shutdown。为了选择一个选项，您可以使用鼠标单击它，或者使用加速键(accelerator key)。

为了断定特定选项的加速键，请仔细地查看所选择的单词。单词中的一个字母有下划线。为了选择这个选项，您可以按下<Alt>键，然后按下这个字母键。例如，假设在单词“Session”中字母“S”有下划线。这意味着 Session 的加速键是<Alt-S>。

提示

加速键是 GUI 的一个标准功能，许多菜单和对话框中都有加速键。例如，在大多数基于 GUI 的程序中，可以通过按下<Alt-F>组合键显示 File 菜单，按<Alt-E>组合键显示 Edit 菜单，按<Alt-H>组合键显示 Help 菜单，等等。

一定要花点时间来学习加速键，因为它们可以使您的工作更方便。您将发现，当您希望进行选择时，按一个简单的组合键要比将一只手从键盘移开来移动鼠标并单击按键方便得多。

下面继续返回登录屏幕，登录屏幕上 4 个选项的使用方式说明如下。

- **Language:** 修改这个特定工作会话所使用的语言。
- **Session:** 选择希望什么类型的工作会话。

通常，不必改变语言或者会话类型，因为默认的语言和会话类型就是您所需的。

- **Reboot:** 重新启动计算机。
- **Shutdown:** 关闭计算机。

一旦输入了用户标识，系统就会提示您输入口令。Linux 然后处理登录过程，启动桌面环境。

当您结束工作时，可以通过从主菜单中选择“Logout”项注销系统。系统将询问您是否确认结束工作会话(根据您所使用的桌面环境，也有可能为您提供是关闭还是重新启动计算机的选项)。

一旦确认希望注销系统，Linux 将终止您的桌面环境，返回到登录屏幕。此时，您可以选择重新登录系统，或者选择重新启动或关闭计算机。

6.4 运行级别

您现在知道 Unix 可以启动成一个基于 GUI 的系统或者一个基于 CLI 的系统。在继续

讨论界面之前,我希望花几分钟时间向您解释一下 Unix 提供这种灵活性的方式。这些概念不仅有趣,而且还向您展示了许多使 Unix 成为如此优秀的一个操作系统的思考和组织方式。

当计算机系统、程序或者设备可以有几种状态时,我们使用术语模式(mode)来指一个特定的状态。例如,我们可能说您可以以文本模式(通过 CLI)或者图形模式(通过 GUI)使用 Unix。

模式是计算领域一个十分基本的概念,计算机人员经常开玩笑地使用这个术语指头脑的状态。例如,一名程序员可能这样告诉另一名程序员:“对不起,昨天我没有来看您。昨天我处于清扫模式中,整个下午一直在打扫房间”。

我提到这一思想的原因在于 Unix 的引导过程是灵活的。这种灵活性是通过使 Unix 拥有以几种不同模式运行的能力来完成的。这些模式称为运行时级别(runtime level),或者简称为运行级别(runlevel)。

运行级别的严格定义多少有些偏于技术:运行级别指允许特定进程组存在的系统软件配置。这是一个不容易理解的定义。因此我们给出一个不太正式的定义,即运行级别指定 Unix 将提供哪些基本的服务。对于不同的运行级别,Unix 提供不同的服务组*。

Unix 系统每次引导时,它都要经历一个复杂的过程。作为该过程的一部分,运行级别就是这时设置的。设置的运行级别控制 Unix 的运行模式。图 6-1 显示了大多数 Linux 发行版使用的运行级别。

运 行 级 别	描 述
0	停机(关机)
1	单用户模式: 命令行
2	非标准化
3	多用户模式: 命令行
4	非标准化
5	多用户模式: GUI
6	重新启动

图 6-1 典型的 Linux 运行级别

在大多数情况下, Linux 默认引导至运行级别 3 或者运行级别 5。如果您的系统设置成引导到运行级别 3,那么您将看到一个基于文本的登录屏幕,您可以按照第 4 章中讨论的方式登录和注销系统。一旦登录系统,您使用的是基本的 CLI。

如果您的系统设置成引导到运行级别 5,那么 Linux 将启动默认的 GUI。您将使用一个图形登录屏幕登录(详见本章前面的描述),而且将使用一个桌面环境来进行工作。

大多数人希望使用桌面环境,因此将运行级别 5 设置成默认的。但是,当系统管理员需要解决服务器(例如 Web 服务器或者电子邮件服务器)问题时,他通常希望使用运行级别 3,这是因为 CLI 允许他快速方便地完成希望的动作(大多数系统管理通过键入命令来完成,而不是在菜单中进行选取)。基于这一原因,桌面系统通常设置成引导到运行级别 5,而服

* Linux 中使用的运行级别系统最初在 System V 中引入。在 BSD 世界中(包括 FreeBSD),并不使用运行级别。相反,系统或者引导为单用户模式,或者引导为多用户模式。因此,本章中有关运行级别的讨论对 Linux 适用,但对 FreeBSD 不适用(有关 System V 和 BSD 的内容,请参考第 2 章)。

务器通常设置成引导到运行级别 3。

运行级别 1 是旧时代的一个延续，那时大多数 Unix 系统由许多用户共享，而由一名系统管理员管理。有时候，系统管理员需要做一些不允许其他人同时登录系统的工作。换句话说，在一个短暂时间内，系统管理员必须将一个多用户的用户系统转换成一个单用户的系统。

为了完成这一点，系统管理员将向所有的用户发送一个通知，告知系统即将关闭(假设 5 分钟后)。当这段时间过去后，他将重新启动 Unix 系统到现在所谓的运行级别 1。这将使系统进入过去所谓的系统维护模式(system maintenance mode)，而现在称为单用户模式(single user mode)。系统管理员在知道了其他人已经不能再登录系统后，就可以进行自己的工作。一旦系统管理员完成自己的工作，他将重新启动系统到多用户模式(运行级别 3 或者 5)，而用户也将再次被允许登录系统。

现在，运行级别 1 已经不太经常使用。这是因为现代的 Unix 系统非常灵活，从而使系统管理员可以在其他用户登录的情况下完成大量的工作——甚至是升级和维护(以前的情况并非如此)。当系统有非常严重的问题时(例如硬盘损坏)，系统管理员才需要将系统引导到运行级别 1。

技术提示

如果您希望使用基本的 CLI 而不是(像大多数人一样)使用 GUI，那么您可以修改系统，将系统设置为默认引导到运行级别 3，而不是运行级别 5。如果使用的是公司或者学校的计算机，那么您可以请求系统管理员*来修改设置。

如果维护自己的计算机，那么您将不得不自己修改设置。这里不准备详细介绍如何修改设置，因为它们已经超出了本书的范围，但是我可以讲一下总体思路。

首先，通过将/etc/inittab 中的 initdefault 的值修改为 3 使系统自动启动到运行级别 3。然后在 rc3.d 目录中检查符号链接，确保 GUI 没有在这一运行级别中自动启动。

提示：如果您在运行级别 3 上使用 CLI，而您希望启动 GUI，那么您可以使用 startx 命令。

在结束这一节之前，还有两个问题需要考虑。但是，我要警告您，不要真的按下面所说的两点进行修改，否则您可能会觉得郁闷。好好地考虑一下：

- (1) 如果将 initdefault 的值设置为 0(Halt)会发生什么情况呢？
- (2) 如果将 initdefault 的值设置为 6(Reboot)又会发生什么情况呢？

6.5 Microsoft Windows 的运行级别

因为有许多人使用 Microsoft Windows，所以将它与 Unix 进行比较有一定的启发意义：Windows 有运行级别吗？(如果不关心 Windows，那么您可以跳过这一节内容。)

这个问题有两种答案。第一种是，Windows 确实有类似于运行级别的启动选项，但是它们并不是运行级别。第二种是，Windows 确实有一个提供运行级别的工具，但是它隐藏得太深，几乎没有人知道它。

* “真正的系统管理员理解运行级别。”——Stephanie Lockwood Childs

首先, Windows 有一个特殊的启动菜单, 可以在系统初始化过程中显示(对于 Windows XP 来说, 可以在系统启动过程中按下<F8>键打开系统启动选项)。该菜单称为 Windows Advanced Options Menu, 它有许多不同的选项, 如图 6-2 所示。

Microsoft Windows 的启动选项
Safe Mode
Safe Mode with Networking
Safe Mode with Command Prompt
Enable Boot Logging
Enable VGA Mode
Last Known Good Configuration
Debugging Mode
Start Windows Normally

图 6-2 Windows XP Pro 版: 启动选项

这些选项确实允许 Windows 引导到不同的模式。但是, 与 Unix 的运行级别不同, Windows 的启动选项是不可配置的, 它们没有提供那么大的灵活性。严格地从技术上讲, Windows 的引导模式更像 Unix 的内核引导选项, 而不像运行级别(如果不理解这一点, 您也不用担心)。

然而, Windows 确实有一个系统可以以运行级别的方式工作, 在这个系统中您可以决定希望运行哪些系统服务。您还可以创建所谓的“硬件配置文件”(右击“我的电脑”, 然后选择“属性”。单击“硬件”标签, 然后单击“硬件配置文件”按钮)。

硬件配置文件允许使用不同的设备配置启动计算机。例如, 对于膝上型电脑, 您可能希望该机器在他人使用和自己使用时采用不同的硬件配置。

这是一个普通常识(至少在您这种在乎自己机器的人之中)。但是大家有所不知的是, 一旦您创建了一个硬件配置文件, 就可以选择为配置文件启用或者禁用哪些系统服务。这与 Unix 的运行级别非常相似。

由于本书不是一本关于 Windows 的书, 所以在此不准备详细讨论这方面的内容。但是, 如果您自己对此有兴趣, 下面将给出 Microsoft Windows XP Professional 的操作步骤。

首先创建一个或者多个配置文件。然后, 从控制面板中双击“管理工具”, 然后是“服务”。右击任意服务, 选择“属性”, 然后单击“登录”标签。这样您将能够为不同的硬件配置文件启用或者禁用该服务。

一个具体的例子就是如果您不希望膝上型电脑连接网络, 那么您可以定义一个特殊的硬件配置文件来实现这一目的。在这种情况下, 您希望禁用所有与网络相关的服务。

6.6 学习使用 GUI

我的猜测是, 即便您从来没有使用过 Unix, 也应该拥有使用 GUI 的经验——来自 Microsoft Windows 或者 Macintosh。但是, 在讨论如何在工作中综合利用 Unix 的 GUI 和

CLI 之前,我希望您对基本的概念有一个完整的了解,因此我准备花几分钟时间来讨论一下那些最重要的思想。

学习使用图形用户界面很容易,但是使用好图形用户界面并不容易。因此即便您是一个熟练的 GUI 用户,最好也花几分钟时间浏览一下本章这一部分的内容。我敢打赌您会发现一些不知道的东西。

尽管不同的桌面环境有许多共同性,但是它们之间也有一些微小而重要的区别,而且也很难编写一套能够适用于所有不同界面的用户指南。另外,GUI 的几乎每一部分都可以定制。您第一次使用某个特定的 GUI 时,将看到一个默认的设置外观,并且以一个特定的方式动作。一旦成为一名熟练的用户,您就可以根据自己的需求和爱好自己定制系统。

在后面 3 节中,我将讨论使用 GUI 必须理解的基本思想。这些内容足够让您开始使用 GUI。在阅读完本章之后,请留出一些时间来阅读随您的桌面环境一起发布的内置帮助信息,特别是查看系统的快捷键。学习如何使用快捷键对于掌握任何 GUI 的使用方法都是十分重要的。

在第 5 章中,已经讨论过几个重要的思想:

- GUI 允许您使用窗口。
- 窗口是屏幕上一个有边界的区域,通常是一个矩形。
- 窗口可以重叠。
- 当需要时,窗口可以调整大小,或者从屏幕上的一个地方移动到另一个地方。

下面继续讨论其他概念。

6.7 鼠标和菜单

在计算机屏幕上,通常有一个可以移动的小图像,这个图像称为**指针(pointer)**。通过使用指点设备——通常是鼠标,可以在屏幕上移动指针。指针的形状可以发生变化,这依赖于您正在做什么以及其在屏幕上的位置。

为了发起一个动作,需要将指针移动到屏幕上的一个特定位置,然后按下按键。鼠标可能有 1 个、2 个或者 3 个按键。**X Window** 被设计为使用 3 键鼠标,这意味着 Unix 桌面环境通常使用一个 3 键鼠标。这些按键称为**左键**、**中间键**和**右键**。

提示

Unix GUI 被设计为使用 3 键鼠标。如果您的鼠标只有 2 个按键,那么您可以同时按下左键和右键来仿真中间键。同时按下鼠标的两个按键称为**和弦(chording)**。另外,如果您的鼠标有一个滑轮,那么按下滑轮也可以仿真中间键。

对于 Macintosh 用户:如果您在 OS X 中使用只有 1 个键的鼠标,那么您可以通过 **Control-Click** 来仿真右键单击,通过 **Option-Click** 来仿真中间键单击。

也就是说,为了实现右键单击,需要按下 **<Control>** 键再单击鼠标按键。为了实现中间键单击,需要按下 **<Option>** 键再单击鼠标按键。

鼠标按键的动作只有两种类型:单击和保持。

如果按下鼠标按钮并松开，则称这种动作为**单击**。如果快速连续地按一个鼠标按钮两次，则称这种动作为**双击(double-click)**。还有一种情况称之为**三击(triple-click)**，即快速地按一个鼠标按钮三次，但是这种情况比较罕见。

当单击鼠标左键时，称为**左键单击(left-click)**。这是最常见的单击类型。同理，还有**右键单击**(较不常见)和**中间键单击**(最不常见)。

当只看到单词“单击(click)”时，通常意味着是左键单击。例如，您可能看到“To pull down the Groatcakes Menu, click on the icon of a groatcake(为了打开 Groatcakes 菜单，可以单击 groatcake 图标)”，这意味着左键单击 groatcake 图标(图标是一个代表某些事情的小图形)。

除了单击，还可以按下鼠标按钮并**保持它**。大多数时候，当希望移动东西时需要保持鼠标按钮。例如，为了移动一个窗口，需要将指针定位到窗口的**标题栏**(窗口顶部显示程序名称的水平区域)上。按下后保持鼠标左键不放并移动鼠标，窗口将跟着鼠标移动。一旦窗口到达希望的位置，就可以释放鼠标按钮将窗口放在这个位置上。

当以这种方式移动对象时，称这种动作为**拖放**。因此，我们可以说为了移动一个窗口，您需要将这个窗口的标题栏拖放到新位置上去。

大多数时间，您可能需要从一个称为**菜单**的列表中进行选择。Unix 中的菜单有两种类型：**下拉菜单**(比较常见)和**弹出式菜单**(较不常见)。

下拉菜单是一种当您单击某个特定单词或者图标时所出现的菜单。例如，大多数窗口在窗口顶部都有一列水平布置的单词(即**菜单栏**)。如果将指针移动到这些单词中的一个上然后单击，那么在这个单词的下面就出现一系列相关选项。然后您就可以在这个列表中进行选择。

最重要的下拉菜单是**窗口操作菜单**，几乎每个窗口中都有这种菜单。为了显示窗口操作菜单，需要单击窗口左边顶部的小图标(在标题栏的左边)。这样做将显示一个与窗口关联的动作列表，最重要的动作包括 Move、Resize、Minimize、Maximize 和 Close。请参见图 6-3(KDE)和图 6-4(Gnome)。

提示

正如本章前面所述，您可以通过按加速键打开菜单。例如，可以按下组合键<Alt-F>打开 File 菜单。

通过使用加速键，可以免除为了移动鼠标并单击按钮而需要将手移开键盘这一不便。

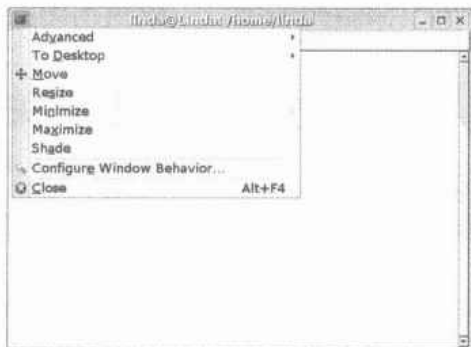


图 6-3 KDE 窗口操作菜单

窗口操作菜单显示了一系列与当前窗口相关的动作。这是 KDE 版本的窗口操作菜单。

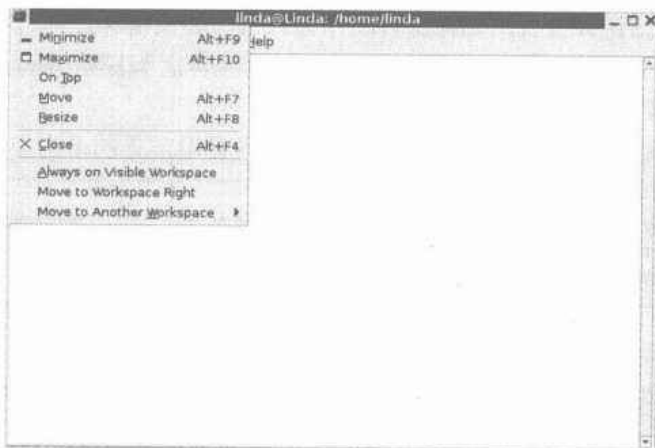


图 6-4 Gnome 窗口操作菜单

窗口操作菜单显示了一系列与当前窗口相关的动作。这是 Gnome 版本的窗口操作菜单。

无论何时当您打开一个菜单时，都要仔细地看看菜单的选项。其中一些选项的边上有一个键名或者一个键组合。这些键称为**快捷键(shortcut key)**。按下快捷键就可以直接选择特定的动作，而且不必打开菜单。

有少数几个快捷键已经标准化，也就是说，在大多数 GUI 的窗口中它们都是相同的，并且值得记住。这类快捷键中最重要一个就是<Alt-F4>。按下这个快捷键将关闭当前窗口。

一定要确保记住了快捷键<Alt-F4>。使用它可以使您在桌面环境中的工作更加流利。如果不知道使用它，那么每次您想关闭窗口时，都不得不使用鼠标，而这种方式比较笨拙和缓慢。就个人而言，我每天要使用<Alt-F4>很多次。

第二种类型的菜单是**弹出式菜单(pop-up menu)**，可以出现在任何地方，但是要一些特定动作发生之后才会出现，该动作通常是鼠标右键单击。按照惯例，右键单击某项将显示所谓的**上下文菜单(context menu)**，也就是说一组与该项自身相关的动作。您可以自己试试这类菜单，即将指针定位到某个您希望的对象上面——例如包含一个程序的窗口——右击并看看会发生什么事情。

6.8 调整大小、最小化、最大化及关闭窗口

正在使用的窗口的大小可以变化以适应您每时每刻的爱好。当您这样做时，我们称其为调整窗口的大小。尽管具体细节可能随各个 GUI 有所不同，但是，窗口大小的调整有两种标准的方法。

第一种方法是使用鼠标修改窗口的边界。将指针移动到希望修改的边界上，然后按住鼠标左键，将边界拖放到一个新位置上。您也可以从窗口的一个角开始拖放。拖放窗口的角可以同时修改两条相邻的边。

第二种方法是使用键盘来移动窗口的边界。打开窗口操作菜单，选择“Resize”，

然后使用箭头键(<Left>、<Right>、<Up>和<Down>)来修改窗口的大小(许多人不知道这一功能)。

同样,您也可以从窗口操作菜单中选择“Move”来移动窗口,然后使用箭头键将窗口移动到希望的地方。当您结束移动时,按<Return>键(或者<Enter>键)即可。

除了修改窗口的大小和移动窗口之外,有时候您可能需要窗口临时消失一会。您不希望关闭窗口(这样将停止程序)。您只希望这个窗口暂时消失,而且在需要它时它能重新出现。

在这种情况下,您要**最小化**(或者 iconify)窗口。这样将使窗口从屏幕的主要部分消失。同时,在**任务栏**(屏幕底部的水平带)上出现窗口的一个小画像(图标)。

当窗口最小化时,窗口中的程序仍在运行。因此,假如您有 7 个窗口,其中 4 个是打开的,3 个窗口最小化到任务栏上。此时,虽然只有 4 个窗口是可见的,但是所有 7 个程序仍在运行中。

一旦窗口最小化了,您可能期望无论何时都能将窗口恢复到它原来的大小和位置。当您这样做时,我们称您在**恢复**窗口。

稍后我将告诉您如何最小化及恢复窗口。在这之前,我希望先提两个有关窗口的动作。第一个是**关闭**动作,可以永久地关闭窗口。这样将停止窗口中正在运行的程序,并使窗口消失。

第二个动作是**最大化**动作,可以最大化一个窗口,将窗口扩展到整个屏幕大小。当您希望只集中于一个任务时,这一动作非常便利。最大化窗口允许您集中在这个任务上,在视觉上不受其他窗口的影响。

那么,如何最小化、恢复、关闭以及最大化窗口呢?方法有若干种。在解释之前,我希望花点时间讨论一下我所说的“窗口控件”。

如果查看一个窗口的右上角,那么您可以看到 3 个小的方框(参见图 6-5)。从右至左,它们分别是 **Close 按钮**(看上去像一个“X”)、**Maximize 按钮**(一个小的矩形)和 **Minimize 按钮**(一个下划线)。

这些按钮的使用非常简单:

- 单击 **Minimize 按钮** 将把窗口最小化到任务栏上。
- 单击 **Close 按钮** 将关闭程序,并停止窗口中正在运行的程序(这和按组合键<Alt-F4>的效果相同)。
- 单击 **Maximize 按钮** 将最大化窗口,即把窗口放大到占据整个屏幕(但是仍然可以看到任务栏)。

但是,您不一定非得单击这些按钮。还有一种方法您可能比较喜欢:从窗口操作菜单中选择相同的动作。通过单击窗口左上角的小图标可以显示该菜单,然后就可以选择“Minimize”、“Maximize”或者“Close”。

一旦程序最小化到任务栏上,您可以通过单击任务栏上该窗口的小画像恢复窗口。这

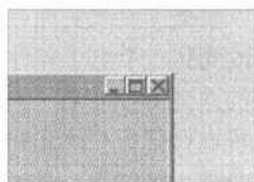


图 6-5 窗口控件

在大多数窗口的右上角,通常可以看到 3 个窗口控件。最右边的控件(“X”)关闭窗口,中间的控件(矩形)最大化窗口,最左边的控件(下划线)将窗口最小化到任务栏上。

样将使窗口扩展回原来的大小和位置。

当最大化窗口时，窗口扩展填充整个屏幕。当这种情况发生时，3 个窗口控件按钮中间的那一个将从最大化(Maximize)按钮变成一个向下还原按钮，即 Unmaximize 按钮(参见图 6-6)。为了将窗口变回到原来的大小和位置，只需单击这个按钮即可。另外，您也可以打开窗口操作菜单，选择相应的选项。

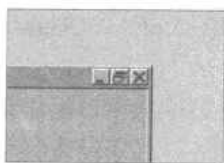


图 6-6 包含 Unmaximize 按钮的窗口

当窗口最大化时，Maximize 按钮就会换成 Unmaximize 按钮。这里共有 3 个窗口控件。右边的控件“X”关闭窗口，中间的控件(重叠的矩形)向下还原窗口，左边的控件(下划线)将窗口最小化到任务栏上。

6.9 控制焦点：任务切换

对于 GUI 来说，我们可以拥有任意数量的窗口，每个窗口都包含有自己的程序。但是，当您按下键盘键、单击或保持鼠标按键时，输入只对一个特定的窗口有效。接受输入的窗口拥有**焦点(focus)**，也就是所谓的**活跃窗口(active window)**。拥有焦点的窗口以某种方式高亮显示。一般情况下，它的标题栏与其他窗口标题栏的颜色不同(参见本章后面的图 6-7)。

您可以根据需要改变拥有焦点的窗口。例如，在课堂上的几分钟之内，您可能由一个网络浏览器改变到一个电子邮件程序，然后又改变到一个字处理程序，接着又改变回浏览器。

焦点的改变方法有若干种。第一种方法是，如果窗口是打开的，那么只需单击它即可。另外，也可以在任务栏上单击该窗口的名称。

不管是哪一种情况，焦点都将传递给您选择的窗口，而且键盘也将连接到该窗口中运行的程序。另外，如果一个窗口被另一个窗口部分遮挡，那么拥有焦点的窗口将在顶层重画，并且完全可见。

另一种改变活跃窗口的方法是使用所谓的**任务切换(task switching)**。窗口中运行的每个程序称为一个**任务**。要查看当前所有正在运行的任务，只需看一下任务栏(通常位于屏幕的底部)即可。每个任务都有自己的小按钮。正如前面所述，如果单击这些按钮中的一个，那么焦点将传递给那个窗口。

这种方式比较好，但是还有一种更方便的方法。按<Alt-Tab>键可以从一个任务切换到另一个任务，这样手就不必再离开键盘了。

当按下<Alt-Tab>键时，GUI 将高亮显示一个特定的任务(您将在屏幕的中央看到一个小的图像，其意义非常明显)。再次按<Alt-Tab>键，下一个任务将高亮显示。

您需要做的就是按下<Alt>键，不停地按<Tab>键，直到定位至您希望的任务。释放这两个键，这个任务的窗口将获得焦点。

提示

为了从一个任务切换到另一个任务，需要不停地按<Alt-Tab>组合键。起初，这种方法可能有点慢，所以下面给出一个小技巧。

用您的左手大拇指按住左<Alt>键不放，用左手的中指不停地按<Tab>键(现在大家可以

试一试)。

为了以相反的顺序浏览任务, 只需用<Alt-Shift-Tab>组合键来替代<Alt-Tab>组合键即可(大家可以试一试)。

一定要花一点时间来练习掌握<Alt-Tab>和<Alt-Shift-Tab>组合键的操作。它们不仅会加快任务的切换, 而且还有助于您改变组织工作环境的方式。

6.10 多桌面/工作空间

当使用 GUI 时, 您所工作的基本空间称为桌面。正如第 5 章中讨论的, 这一名称是一个隐喻。就这一点而论, 在桌面上打开的窗口就像一个真实的桌子上的纸张一样。

以上隐喻的含义并不深刻, 因此我们不再采用它。相反, 我希望您认为桌面是一个抽象环境, 在这个环境中您可以组织您的工作。桌面的特征已经远远超出了屏幕的物理实物所对应的范围。如果您要掌握 Unix 工作环境*的话, 那么理解这些特征非常重要。

这里最重要的思想就是您可以拥有不止一个桌面, 每个桌面都拥有自己的背景、自己的窗口、自己的任务栏等。从一个桌面切换到另一个桌面非常简单, 而且当您这样做时, 会感觉到正在切换到一个全新的系统。

名称含义

桌面、工作空间

在 GUI 中, 桌面是基本的工作环境。桌面包含背景、窗口、任务栏等。

大多数桌面环境允许使用多个桌面。这样将带来一个优点, 即能够创建多个外观和感觉上相像的图形工作环境。

但是, 名称“桌面”可能使人迷惑, 因为它通常还用于指桌面环境本身, 也就是整体上的 GUI。基于这一原因, 您经常可以看到将桌面称为工作空间(workspace), 这样就更切合实际了。

例如, 当使用 KDE 时, 您使用“桌面”。当使用 Gnome 时, 您使用“工作空间”。无论怎么称呼, 它们都指同一件事情, 而且抛开细小的细节, 它们还以相同的方式工作。

从一个桌面/工作空间切换到另一个桌面/工作空间的方式有两种, 您可以使用鼠标或者键盘。

在使用鼠标切换桌面/工作空间时, 请查看屏幕底部任务栏的附近。您将看到一组小的方块, 一个方块代表一个桌面。例如, 如果您有 4 个桌面, 那么这里将有 4 个方块, 如图 6-7 所示(在本章后面)。为了切换到某个桌面, 只需单击这个桌面的方块即可。尽管这听起来有点含糊, 但是一旦您掌握了它事情就会变得简单。大家只需多加练习即可。

在使用键盘切换桌面/工作空间时, 可以使用快捷键。根据桌面环境的不同, 快捷键也有所不同, 因此最好的办法就是查看“Shortcut Keys”文档(查找并单击 Help 图标)。

* 在大约 2000 年之前编写的 *Yoga Sutras* 中, 古代的圣人 Patañjali 提出的就是这一概念。在该书第 2 篇, 第 21 节中, 他写道(用原始的梵文书写): Tadarthah eva drsyasya atma。

这句话可以翻译为: “Unix 桌面的本质和智能主要是为用户的真实目标服务, 解放用户。”

对于 KDE, 桌面快捷键是<Ctrl-Tab>和<Ctrl-Shift-Tab>。

对于 Gnome, 桌面快捷键是<Ctrl-Alt-Left>、<Ctrl-Alt-Right>、<Ctrl-Alt-Up>和<Ctrl-Alt-Down>(也就是说, 在按住<Ctrl>和<Alt>键的同时按一个箭头键)。

使用桌面快捷键是如此的简单, 或许您会很快地掌握它, 从而不必再使用鼠标从一个桌面切换到另一个桌面。

一般而言, 桌面环境默认提供 4 个不同的桌面。但是, 如果您喜欢, 可以添加任意多的桌面。为了添加桌面, 只需在屏幕底部的小方块中单击鼠标右键。这将弹出一个上下文菜单。然后您就会看到一个类似于“Preferences”或者“Desktop Configuration”的菜单项。

在结束本节内容之前, 我希望提出一个本章末尾还会再提及的思想。到目前为止, 我已经展示了 Unix 工作环境如何使同时处理多个事情成为可能。在每个桌面中, 您可以打开多个窗口, 每个窗口运行一个独立的程序。您可以拥有多个桌面, 而每个桌面都拥有自己的一组窗口。另外, 通过使用快捷键, 可以方便地从一个桌面切换到另一个桌面, 而且在一个桌面内, 还可以从一个窗口方便地切换到另一个窗口。

我希望您思考下面的问题: 给定所有这些资源, 什么才是您组织工作的最佳方式呢? 不久您将会明白, 这一问题的答案对您的 Unix 体验非常重要。

提示

在组织工作时, 可以将窗口从一个桌面移动到另一个桌面。

右击窗口的标题栏。这样将弹出一个菜单, 让您将窗口移动到选定的桌面上。

6.11 终端窗口

在第 3 章中, 我们讨论了终端, 以及以前如何使用它们访问多用户 Unix 系统(通过使用 CLI 或命令行界面)。现在, 我们讨论在不使用实际终端的情况下如何访问 Unix 系统。实际上, 我们运行一个仿真(充当)终端的程序。当在 GUI 下运行这样一个程序时, 该程序仿真一个 X 终端(第 3 章末讨论过的图形终端)。

前面已经解释过, Unix 的大多数功能要通过 CLI 才能发挥出来。这意味着在您阅读本书的过程中, 要做的大多数事情就是在 shell 提示处输入命令(一条接一条), 以及使用基于文本的程序。为了这样做, 您需要访问终端。

很明显, 我们不使用真正的终端。实际上, 我们使用的是终端仿真器。那么问题是我们如何访问终端仿真器呢?

对于现代的 Unix 系统来说, 方法有两种, 而且两种方法都重要。我们现在就讨论其中一种(终端窗口), 另一种方法(虚拟控制台)在下一节讨论。

在桌面环境中, 所有的工作都是在窗口中完成的, 因此只有在窗口中运行终端仿真程序才有意义。

这样做非常简单。所有的桌面环境都提供一种启动终端程序的简单方法。只需打开主菜单就可以看到它。例如, 在 Gnome 中, 如果查看 System Tools 子菜单, 您就会看到“Terminal”。在 KDE 中, 如果查看 System 子菜单, 您就会找到两个这样的程序“Terminal”

和“Konsole”(稍后将解释 KDE 中为什么有两个这样的程序)。

当启动这样的程序时，屏幕上将出现一个窗口。在这个窗口中，有一个标准的 CLI。作为示例，请看一看图 6-7，这个图中有两个终端窗口。

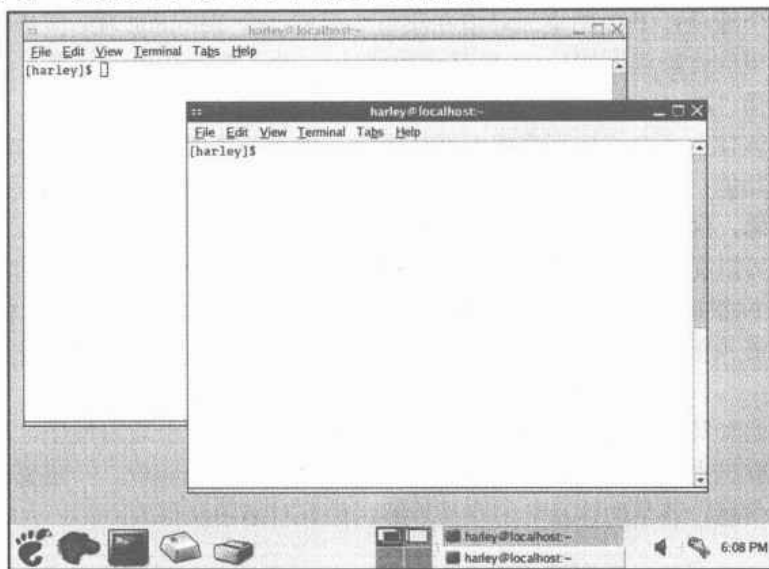


图 6-7 多个终端窗口

在桌面环境中，您可以打开任意数量的终端窗口。在终端窗口中，可以使用标准的 Unix CLI(命令行界面)输入命令。在这个例子中，打开了两个终端窗口。拥有焦点的活跃窗口是标题栏颜色较深的窗口。

第 4 章中讲过，shell 是命令处理器。它显示一个提示并等待输入命令。然后它处理命令。当命令完成时，shell 显示另一个提示，并等待下一条命令。

在我们的例子中，在第一个 shell 提示处，输入了命令 **date**(显示当前时间和日期)。shell 处理这条命令，然后显示另一个提示。

因为终端仿真器运行在窗口中，而且您可以拥有任意数量的窗口，所以您可以同时运行多个终端。尽管您可能奇怪为什么您希望这样做，但是这一时刻很快就会到来——一旦您成为一名有经验的 Unix 用户，需要同时让多个命令行程序工作(每个程序都位于自己的终端窗口中)，您就会觉得这种事情再平常不过。

提示

大多数桌面环境都有一组图标，可以用来启动经常使用的程序(这些图标位于屏幕的顶部或者底部)。如果右击这一区域，将出现一个弹出式菜单，允许您添加或者删除程序(通过图标)。

我的建议是添加您喜爱的终端程序。采用这种方式，当您需要时，只需单击一个图标，就可以打开一个新的终端窗口。

这一点非常重要，建议您现在就花一点时间尝试一下。同时，您还可以通过删除不再使用的程序的图标来减少混乱。

正如前面所述，KDE 有两个终端程序可供选择。实际上，对仿真终端的需求已经有很

长一段时间了，所以产生了许多不同的终端程序，而且大多数桌面环境都提供多种终端程序供选择。

一些桌面环境允许直接从菜单中选择希望的终端程序(KDE 就是这种情况，在其菜单中可以看到两个选项)。其他的桌面环境在菜单中只有一个 Terminal 项，但是可以将它改变为指向任意希望的终端程序(Gnome 就是这种情况)。为了查看选项，只需在 Terminal 项上右击，然后选择“Properties”即可。

那么您应该使用哪个终端程序呢？严格地讲，这无关紧要，因为所有的终端程序都提供相同的基本功能：在一个窗口中提供 CLI。但是，如果您喜欢别出心裁，那么 Konsole 程序特别适合您，因为它拥有许多其他终端程序中没有的有用特性。

例如，您可以在同一个窗口中运行多个 CLI 会话，每个会话拥有自己的标签。然后您可以通过单击标签或者使用快捷键<Shift-Right>和<Shift-Left>(也就是说按住<Shift>键的同时按下向右箭头键或向左箭头键)从一个会话切换到另一个会话。您将会发现这些组合键特别方便。

我的建议是如果系统上有的话，就使用 Konsole。它是一个通用程序，即使没有使用 KDE，您的计算机上也可能安装有 Konsole。一旦开始使用 Konsole，一定要花一些时间阅读内置的帮助信息。该帮助信息非常容易理解，而且对您帮助很大。

名称含义

xterm、xvt、Konsole

当 X Window 刚被开发出来时，它提供了一个叫做 **xterm** 的终端程序，该程序仿真 VAXstation 100(VS100)图形终端(有关图形终端的讨论，请参见第 3 章)。最初版本的 **xterm** 于 1984 年夏天由一名麻省理工学院的学生 Mark Vandevoorde 开发。

(非常有趣的是，Vandevoorde 开发的 **xterm** 是一个独立的终端仿真器，它与 X 没有关系。但是，在极短的时间内，**xterm** 进行了彻底改动，被移植到新的 X Window 系统上，从那时起 X Window 系统就一直提供它。基于这一原因，人们为 X Window 终端保留了 **xterm** 这一名称。)

现代版本的 **xterm** 既可以仿真基于文本的终端(旧的 DEC VT-102，与 VT-100 非常相似)，也可以仿真图形终端(Tektronix 4014)。在很长一段时间内，**xterm** 是 X 中的主要终端程序。

1992 年，Kent 大学的 John Bovey 编写了 **xvt**，一个替代 **xterm** 的新终端仿真器。尽管 **xvt** 或多或少地与 **xterm** 兼容，但是这个新程序的速度较快并且需要较少的内存(在那个时代这是一个大问题)。一个原因就是它只是一个基本的 VT-100 仿真器，它并不支持 Tektronix 4014(因此，名称 **xvt** 代表 X Window VT-100 仿真器)。

自 20 世纪 90 年代开始，程序员们编写了许多基于 X Window 的终端程序，而且几乎所有的程序都或者基于 **xterm**，或者基于 **xvt**。1998 年，一名德国程序员 Lars Doelle 发布了 Konsole，这是一个全新的终端程序。Konsole 是 KDE 项目的一部分，而且多年以来，它一直在不停地更新，因此，现在它是全世界功能最强大的终端仿真器之一。

顺便说一下，Konsole 是“console”的德语单词。这是一个非常奇妙的巧合，因为正如第 5 章中所述，KDE 程序的命名习惯以字母 K 开头。

6.12 虚拟控制台

1980 年, 我们前往一流的 East Coast 大学, 访问校园中的最重要教授(Most Important Professor, MIP)。MIP 拥有一个神奇的办公室, 里面放着一张大桌子和 7 台不同的终端, 所有这些终端都连接到一个功能强大的 Unix 系统上(参见图 6-8)。

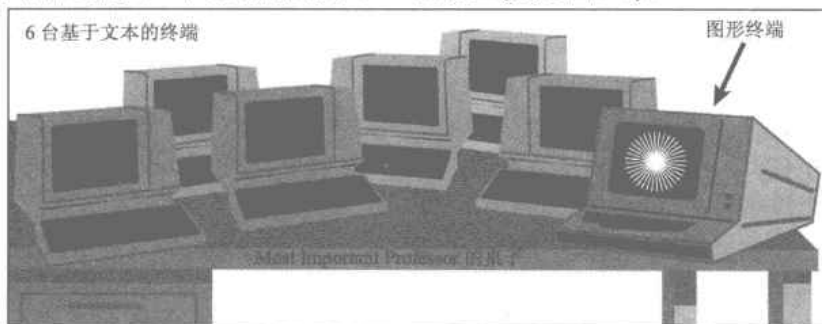


图 6-8 一名用户多台终端

在 20 世纪 80 年代, 这是大学中“最重要教授”的桌子。MIP 是如此的重要, 他一个人就使用 7 台终端: 6 台基于文本的终端和 1 台图形终端(最右边的那个)。现在, 当您使用自己的 Linux 计算机时, 您拥有更出色的装备: 7 个内置的虚拟控制台(详情请参见正文)。

MIP 向我们展示了他的办公室。“这些终端中有 6 台是字符设备, 提供标准的命令行界面,” 他解释道, “实际上, 这台特殊的终端是控制台。”

等待了一秒钟, 我们问: “难道控制台不是在计算机附近, 或者至少应该在系统管理员的办公室吗?”

“大多数情况下是这样的,” MIP 说, “但是我是一位非常重要的人物, 所以为我准备了 6 台字符终端。我经常同时处理不止一个项目, 因此将所有的终端让我自己使用, 这样就比较方便。如果系统出现了问题, 我可以让系统管理员来我的办公室使用控制台。”

“您是多么的和蔼可亲,” 我们说, “那您真得使用所有 6 台终端吗?”

“的确如此, 我使用所有 6 台终端。我是重要人物。我需要许多终端。但是, 我有时候也让秘书*使用一台。”

最后, MIP 指着第 7 台终端, 即最边上的那一台。“这是一台图形终端, 非常特殊。”他说, “不错, 这样的终端上只运行有限数量的程序, 因为我如此重要, 所以必须有一台。”

我们向前跨越 25 年。您刚刚阅读了“最重要教授”的故事, 您可能会想: “假如我自己有 7 台终端。这是不是太神奇了? 我希望我是重要的人物。”

* 秘书是哺乳动物版的个人数字助理。从 20 世纪 30 年代到 20 世纪 80 年代末, 秘书在大学环境中非常盛行。

与 PDA 相比, 大学秘书不可靠、难以控制、不可更新, 而且还经常有记忆问题。此外, 因为劳工许可证限制, 在 20 世纪 80 年代和 90 年代, 许多秘书不能被解雇, 即使他们的工作很差。

1996 年, 秘书被列为濒临灭绝的职业种类, 而且自 1998 年起, 就可以认为他们已经灭绝。最后一位有记录的大学秘书位于美国 Fargo 市 North Dakota 大学, 时间是 1998 年末。但是, 这一记录没有得到确认, 而且大多数专家认为“秘书”现在实际上已经变成资料管理员了(也位于濒临灭绝的职业列表中)。

实际上,如果拥有自己的 Unix 计算机,那么您就是重要人物。所有现代的 Unix 系统都允许同时使用多个终端,其中有一个终端是图形终端。

此外,您也不必在自己的桌面上混乱地堆着不止一台显示器和键盘。Unix 将为每个终端使用同一个显示器和键盘。您所需做的全部事情就是按快捷键从一个终端切换到另一个终端。

下面是 Linux 系统中多个终端的运转方式(其他系统与此相似)。

当启动 Linux 时,GUI(也就是桌面环境)自动启动。您可能不知道的是,实际上 Linux 同时为您启动了 7 个不同的终端仿真程序。它们称为虚拟控制台(遗憾的是,该名称易于引起误解,我将在下一节中讨论这一点)。

虚拟控制台#1~6 都是全屏、基于文本的终端,用来使用 CLI。虚拟控制台#7 是图形终端,用来运行 GUI。实际上,当桌面管理器启动时,您所看到的实际上就是虚拟控制台#7。其他 6 个虚拟控制台此时都不可见。

为了从一个虚拟控制台切换到另一个虚拟控制台,您需要按特殊的组合键。对于虚拟控制台#1,按组合键<Ctrl-Alt-F1>(也就是说,在按下<Ctrl>键和<Alt>键的同时按下<F1>键)。对于虚拟控制台#2,按组合键<Ctrl-Alt-F2>;对于虚拟控制台#3,按组合键<Ctrl-Alt-F3>;依此类推。

当按下这些组合键中的一个时,您将立即看到一个全屏的 CLI 终端。您可以根据需要使用 6 个这样的终端。为了返回到 GUI(桌面环境)中,只需按组合键<Ctrl-Alt-F7>即可(记住,第 7 个虚拟控制台是图形终端)。

提示

从一个虚拟控制台切换到另一个虚拟控制台的快捷键实际上是<Alt-F1>(终端#1)至<Alt-F7>(终端#7)。但是,在大多数 GUI 中,这些键有其他的用途,因此还必须按下<Ctrl>键。

例如,假如您正在使用 GUI(终端#7)。为了切换到终端#3,您需要按组合键<Ctrl-Alt-F3>。

假如您正在使用一个 CLI,为了切换到终端#4,可以按<Alt-F4>;为了切换到终端#1,可以按<Alt-F1>;依此类推。为了切换回终端#7(GUI),则需要按<Alt-F7>。

那么为什么您希望使用<Alt>自己而不是<Ctrl-Alt>呢?因为这样更简单更快速,而 Unix 用户希望事情尽可能简单快速。

当从一个虚拟控制台切换走时,在这个虚拟控制台中运行的程序仍然继续运行。例如,您在终端#4 中启动了一个执行很长的任务(例如编译一个大型程序或者从 Internet 上下载大量的文件)的程序。然后您按下<Alt-F5>切换到终端#5 进行其他的工作。过一段时间后,您可以按<Alt-F4>返回到终端#4,查看程序的运行情况。

这里又出现了一个问题,即为什么要在 6 个基于文本的虚拟控制台之间来回切换呢?为什么不在 GUI 中使用终端窗口和多个桌面呢?原因有多方面。

首先,基于文本的虚拟控制台使用整个屏幕,并且使用一种等宽字体显示字符。整体来讲,这是一种特别令人满意的 CLI 使用方式,要比 GUI 中的终端窗口(即便是最大化的窗口)好许多(大家可以试一试)。

其次,按<Alt-F1>、<Alt-F2>等组合键从一个命令行切换到另一个命令行是一种很酷的体验(大家可以再试一试)。

第三,如果桌面环境发生严重的问题,那么切换到虚拟控制台去解决问题将十分方便。例如,如果 GUI 失去响应,您通常可以按<Ctrl-Alt-F1>进入另一个终端。然后您可以登录来修复问题,或者重新启动系统。

为了完成我们的讨论,下面解释一下为什么要讲述“最重要教授”的故事。原因有两方面。

首先,我希望您意识到现代的 Unix 功能实际上有多强大。如果使用 1980 年的 Unix,那么您将不得不共享终端,通常还要去一个终端室排队等候。如果您试图拥有自己的个人终端,那么您需要是一名重量级的人物。

事实是在 1980 年,计算机设备如此的昂贵,即便是大学中最重要的人物也不能在自己的办公室中放几台终端,更不用说 7 台了。然而现在,当您在 PC 上使用 Linux 时,您在自己的桌面上就可以拥有这么多的终端(实际上,FreeBSD 提供 8 台终端)。

其次,我希望您注意多个虚拟控制台这一思想,实际上是四分之一世纪之前硬件实物的一个软件实现。以前的计算范例现在还有多少个仍然如此有价值呢?到目前为止,在本书中,我们已经遇到了两个:客户端和服务系统(第 3 章)以及灵活可扩展的 X Window 设计(第 5 章)。除这两个之外,现在可以添加另一个基本原则。

Unix 基于这样一种思想,即使用终端访问主机计算机,这一思想是如此的强大,以至于实际终端已经消失很久了,但是由它们启发而开发的系统仍然在茁壮成长。

名称含义

虚拟

在计算机领域,术语“虚拟”用来描述一些由软件仿真的对象。例如,本章中我们讨论的虚拟控制台就不是真实的,从该意义上讲,它们没有一个独立的物理存在。它们存在只是因为它们是由软件创建的。

一个更重要的例子就是虚拟内存这一思想,虚拟内存是一种在计算机中模拟出比物理内存更多内存的方法。例如,如果一台计算机有 2GB 的内存,那么操作系统可以通过用硬盘存放超量的数据的方式来模拟内存,从而提供 4GB 的虚拟内存(顺便说一下,Unix 系统使用虚拟内存)。

术语“虚拟”来源于光学,光学是物理学的一个分支。假设您在镜子前面看一支蜡烛。当您看实际的蜡烛时,物理学家说您正在看“真实的”图像。当您看镜子中的蜡烛时,您就在看“虚拟的”图像。

6.13 唯一的控制台

在第 3 章中,我们讨论了控制台。这个术语可能使人有点迷惑,因此我们需要花点时间了解它。

以前,大多数 Unix 计算机都连接着许多终端。当有人希望使用系统时,他需要寻找一台没有人使用的终端进行登录。

其中有一台终端非常特殊,这台终端就是控制台,这台终端之所以特殊是因为系统管理员使用它来管理系统。在大多数情况下,控制台存放在一个上锁的房间内,其位置或者

紧挨着主机计算机，或者在系统管理员的办公室内。

从物理上讲，控制台和其他终端一样，但是有两件事情使它们有所不同。首先，当 Unix 需要显示一个非常重要的消息时，这个消息就会发送到控制台上。

其次，当系统引导到单用户模式(为了维护或者解决问题)时，只有控制台是激活的。通过这种方式，其他用户就不能登录系统，直到系统重新启动到多用户模式中。

现在，“控制台”通常用作终端的一个同义词，这容易使人迷惑。例如，刚刚讨论的虚拟控制台就不是真正的控制台。称它们为“虚拟终端”会更恰当一些。

本章前面提到的 Konsole 终端仿真程序也是这种情况。该程序由一名德国程序员编写，而且 Konsole 是德语版的“console(控制台)”。然而，这一名称不准确。当运行 Konsole 时，得到的是一个终端窗口，不是控制台。

6.14 选择与插入

当使用多个窗口工作时，每个窗口中都包含有自己的程序，所以有时候需要从一个窗口向另一个窗口复制或者移动信息。如果您使用的是 Microsoft Windows 或者 Macintosh，那么您应该对复制和粘贴已经非常熟悉。

对于 Unix 来说，有两种不同的方法来复制数据。第一，X Window 允许从一个窗口到另一个窗口或者在同一个窗口内选择和插入文本。第二，许多基于 GUI 的程序支持 Windows 类型的复制/粘贴功能(<Ctrl-C>、<Ctrl-V>等)。

这两种方法完全独立，并且工作方式不同，因此您需要同时学习如何使用这两种方法。我们首先从 X Window 系统开始，因为它相对比较简单。

X Window 的选择/插入功能只针对文本，也就是字符。首先，您用鼠标选择一些文本(一会儿进一步解释)。然后移动鼠标到希望插入文本的地方，按下鼠标中间键。只要您一单击鼠标中间键，所选择的文本就会插入。如果鼠标没有中间键，则可以同时单击鼠标的左键和右键。

在选择文本时，首先将鼠标指针定位于希望选择文本的开始处。然后按住鼠标左键不放，在文本上拖动指针。在拖动指针的过程中，选中的文本会高亮显示(您可以打开一个终端自己试一试)。

另外，您还可以以另外两种方式选择文本。如果双击鼠标，则会选中一个单词；如果三击鼠标，则会选中整行文本(试一下。这些操作需要多加练习)。

尽管在阅读这一部分内容时，这些说明听起来可能有点复杂，但是一旦掌握了它们，选择和插入文本实际上相当快速容易。如果您坐在自己计算机的前面，那么我建议现在就花一点时间，打开两个窗口，练习一下文本的选择和插入。如果可能，还可以让他人给您示范一下。

提示

因为是 X Window 提供的选择/插入功能，所以只能在 GUI 中使用它。在虚拟控制台中无法使用这一功能。

但是，有一个叫 **gpm** 的 Linux 程序将这一功能扩展到基于文本的虚拟控制台(#1~6)。

至于 FreeBSD, 也有一个相似的程序 `moused`。

如果您的系统上安装有 `gpm` 或 `moused`(通常情况下默认安装), 那么您可以在基于文本的虚拟控制台上以与 GUI 相同的方式选择和插入文本。两者之间的唯一区别就是在插入文本时, 虚拟控制台中单击的是鼠标右键, 而不是鼠标的中间键。

6.15 复制与粘贴

正如上一节所解释的, 您可以使用 X Window 的选择/插入功能在一个窗口中或者两个窗口之间复制文本。除了选择和插入外, 许多(但并不是全部)基于 GUI 的程序还支持一种完全不同的方式, 即复制和粘贴。

Unix 的复制和粘贴功能的工作方式与 Microsoft Windows 相同, 甚至还使用相同的键和菜单。但是, 与 Windows 不同, 并不是所有基于 GUI 的 Unix 程序都使用复制/粘贴。您必须自己确定自己的程序是否支持这一功能。

在详细介绍细节之前, 先给出一个简短的摘要: 首先, 您需要将数据“复制”或者“剪切”到“剪贴板”中。然后, 再将数据从剪贴板中“粘贴”到任何有意义的地方。大多数时间复制的数据是文本, 也就是字符。但是, 您也可以复制图形, 只要您使用的程序支持这样做。

为了进行复制/粘贴操作, 需要理解下面的 4 个基本思想:

- 剪贴板是一个看不见的存储区。
- 当把数据复制到剪贴板时, 原始数据不会发生变化。
- 当把数据剪切到剪贴板时, 原始数据将被删除。
- 为了从剪贴板中复制数据, 需要将数据粘贴到窗口中。

因此, 为了将数据从一个位置复制到另一位置, 需要复制和粘贴操作。为了将数据从一个位置移动到另一位置, 需要剪切和粘贴操作。

无论何时, 当您复制或者剪切数据时, 剪贴板中的当前内容都将被替换, 您再也无法取回它们。例如, 假如剪贴板中包含有 50 段文本, 然后您复制一个单独的字符到剪贴板中。那么, 现在剪贴板中只包含这一个字符, 而前面的 50 段文本不见了。

另一方面, 当粘贴数据时, 剪贴板中的内容不会发生变化。这意味着剪贴板中的内容将保持不变, 直到您执行另一次复制或者剪切操作为止。这样将允许您多次粘贴相同的数据(当然, 在重新启动或者关闭系统时, 剪贴板中的内容就会丢失)。

那么如何进行复制、剪切和粘贴呢? 首先, 您必须选择希望放到剪贴板中的数据。这一步操作有两种实现方法。

第一, 您可以使用鼠标。将鼠标指针定位到希望复制或者剪切的数据的开始处, 然后按住鼠标左键, 在文本上拖动鼠标指针。在拖动鼠标指针时, 选中的文本就会高亮显示。

另外, 如果需要选择的数据是可以键入的, 那么您可以通过键盘进行选择。将光标定位到希望复制的文本的开始处, 按住<Shift>键不放。在按住<Shift>键的同时, 使用光标控制键移动光标, 所经过的文本将被选中。

一旦选取了数据, 就可以用 3 种不同的方式复制或者剪切数据。您可以通过<Ctrl-C>复制数据或者<Ctrl-X>剪切数据, 也可以通过单击鼠标右键弹出一个菜单并选择“Copy”

或者“Cut”来复制或剪切数据，还可以通过打开 Edit 菜单选择“Copy”或者“Cut”来复制或剪切数据。

同样，您可以用 3 种不同的方式来粘贴数据：将光标移动到希望粘贴数据的地方按<Ctrl-V>粘贴数据；或者单击鼠标右键弹出一个菜单选择“Paste”来粘贴数据；又或者打开 Edit 菜单，然后选择“Paste”来粘贴数据。

提示

如果在剪切或者粘贴过程中犯了错误，那么您可以通过<Ctrl-Z>或者选择菜单中的“Undo”取消上一次操作。

对于一些程序，按<Ctrl-Z>多次可以取消最近的多次操作。

6.16 以超级用户工作：su

为了保证系统的安全，Unix 被设计成每个用户标识拥有一组有限的权限。例如，您可以删除自己的文件，但是不能删除其他人的文件，除非他人已授予您明确的权限。同理，作为一名普通用户，您不能修改或者运行影响系统整体性的程序。

在第 4 章中，我们已经讨论了一个观点：有时候，为了获得超出普通用户所拥有的权限，系统管理员有必要以超级用户的身份登录。例如，系统管理员可能需要在系统中添加一个新用户、修改某人的口令、升级或者安装软件等。

前面已经解释过，Unix 系统中有一个特殊的用户标识 **root**，它就拥有这样的权限。当您以 **root** 登录时，您就是超级用户，您可以做您想做的任何事情(很明显，**root** 的口令是一个非常重要的秘密)。

如果您使用的是一个共享系统，例如学校或者单位的系统，那么会有人负责系统的管理，因此您不用担心。例如，如果您忘记了自己的口令，可以请求他人帮忙。但是，当您使用自己的 Unix 系统时，例如一台运行 Linux 或者 FreeBSD 的 PC，您自己就是系统管理员，因此您必须知道怎样以超级用户的身份工作。

为了成为超级用户，您需要超级用户的口令(通常称为 **root** 口令)。如果在自己的计算机上安装 Unix 系统，那么在安装过程中系统将要求您选择 **root** 口令。

提示

一些 Linux 发行版设置成在安装过程中不创建 **root** 用户标识。在这种情况下，可以使用普通用户的口令作为超级用户口令。

提示

在任意一个 Unix 系统上，**root** 口令都是您所拥有的最重要的信息。因此，您要记住它，而不是把它写下来。但是，如果由于某种原因，您碰巧忘记了自己计算机的 **root** 口令，那么这就是一个很大很大的问题。

以前，除了重新安装 Unix 系统之外，没有其他可供选择的方法。但是对于现代系统而言，还有一种方法来解决这一问题。详情请参见“附录 E：忘记 **root** 口令的处理方法”。

一旦知道了 **root** 口令，有两种方法可以成为超级用户。第一种，在登录提示处，您可以以 **root** 登录。当系统初次启动或者使用虚拟控制台时，您可以以 **root** 登录。

另外一种，如果您已经作为一名普通用户登录系统，那么您可以使用命令 **su**(substitute userid, 替换用户标识)变成超级用户。

su 命令允许您临时变成另一个用户标识。为此，只需在 **su** 命令后输入新的用户标识即可。下面举例说明。

假设您现在已经以 **harley** 登录系统，下面是一种可能的 shell 提示：

```
[harley]$
```

(在这个例子中，我们使用的是 **Bash shell**。已经将 shell 提示设置为显示当前的用户标识。我们将在第 13 章中详细讨论 shell 提示。)

现在您输入命令 **su** 将用户标识变换到 **weedly**。然后系统提示您输入 **weedly** 的口令。一旦您输入了 **weedly** 的口令，当前 shell 就被挂起，系统将为 **weedly** 启动一个新 shell。

```
[harley]$ su weedly
Password:
[weedly]$
```

当您结束 **weedly** 的工作时，您需要做的就是结束当前 shell。输入 **exit** 命令就可以结束当前 shell。一旦结束了新 shell，您就会自动返回到原来的 shell，即用户标识 **harley**。

```
[weedly]$ exit
[harley]$
```

尽管您可以使用 **su** 命令变换到任何一个用户标识(如果拥有口令的话)，但是 **su** 命令的主要用途还是变换到超级用户。在详细介绍之前，我希望先解释一件事情。

无论何时，当您登录时，Unix 都会运行一些特定的命令为您的用户标识建立一个具体的环境(本书后面，当讨论 shell 时，我将示范如何定制这一过程)。假设您以 **harley** 登录后，输入了下述命令：

```
su weedly
```

您将用户标识变换成 **weedly**，但是仍然在 **harley** 的环境下工作。如果希望同时变换用户标识和环境，则需要在 **su** 命令名称之后键入一个-(连字符)。注意，连字符的两边都有一个空格。

```
su - weedly
```

现在您在 **weedly** 的环境中以 **weedly** 的名义工作了。当您输入 **exit** 命令时，将返回到 **harley** 环境中以 **harley** 的名义工作。

我知道这听起来有点神秘，但是，我向您保证，总有一天，它的价值会体现出来。现在之所以提它是因为我准备向您示范如何使用 **su** 命令变换成超级用户。当您这样做时，您要在超级用户的环境中工作，而不是在自己的环境中工作，这一点很重要。

为了变换成超级用户，需要使用命令 **su**，后面跟用户标识 **root**。记住不要落下连字符。

```
su - root
```

现在系统向您询问一个口令。一旦您输入了口令，当前的 shell 将被挂起，系统将启动一个新的 shell。在新的 shell 中，您的用户标识将是 **root**，而且您将拥有完全的超级用户权限。

```
[harley]$ su - root
Password:
#
```

注意 shell 提示符已经变成了 #。正如第 4 章中解释的，这表示您已经成为超级用户。

当您结束需要成为超级用户才能完成的工作时，键入 **exit** 命令，您将返回到自己的旧 shell 中：

```
# exit
[harley]$
```

为了方便起见，如果 **su** 命令没有指定用户标识，那么默认的用户标识是 **root**，大多数时候您需要的就是这种方式。因此，下述两个命令是等效的。

```
su -
su - root
```

最后强调一句，如果您的系统由许多人共享，那么在作为超级用户时您必须加倍小心。这一方面有许多内容可以讲，但是所有的内容都可以概括成两个基本规则：

- (1) 尊重其他人的隐私。
- (2) 在键入之前认真思考一下。

重要提示

当以 **root** 登录时，您可以做任何想做的事情，但是这样可能导致错误。防止偶然产生问题(或者灾难!)的最好方法就是只在必要时临时以超级用户登录。

例如，在我的系统上，我一般以 **harley** 登录系统。当需要执行系统管理的任务时，我使用 **su** 命令变换成超级用户，直到完成超级用户的工作。然后我立即返回到 **harley**。在完成超级用户的工作后，已经没有必要再继续以 **root** 登录，因为无论何时当我需要时都可以再变换成超级用户。

换句话说，就是在大多数时间里要做 Clark Kent(即美剧《超人前传》中的人物，译者注)，而不是超人(Superman)。

6.17 以超级用户执行一条单独的命令：sudo

正如上一节所述，作为超级用户的时间太长会非常危险：您可能不小心键入了导致故障的内容。另外，这样还有一个潜在的问题。

假如以 **root** 登录后，您碰巧觉得饿了。然后您离开计算机，准备去拿一些热奶油细燕麦粉蛋糕。在离开时，您出乎意料地与他人争论起来，争论南半球沉下去的水是否比北半球沉下去的水以相反方向流动要多^{*}。当您回来时，发现在您离开时，有一个聪明的家伙获

^{*} 事实并非如此。

得了对您机器的控制，充当超级用户删除了数百个文件。

当然，无论以什么用户标识登录，您都不应该离开自己的计算机。这就像离开自己没有上锁的汽车，并将钥匙藏在座位下面一样。但是，以 **root** 登录时离开自己的计算机就好比离开自己没有上锁的汽车时，发动机发动着而且门还开着。

防范这些不经意间错误的最好方法就是使用 **sudo** 命令。

sudo 命令允许您以另一个用户标识执行一条单独的命令(名称 **sudo** 的意思是“替换当前用户标识，然后做一些事情”)。和 **su** 命令一样，**sudo** 命令的默认用户标识为 **root**。因此，为了以超级用户执行一条具体的命令，只需键入 **sudo**，后面跟着这条命令即可：

```
sudo command
```

例如，为了以 **root** 用户标识执行 **id** 命令，可以输入：

```
sudo id
```

(**id** 命令显示您的当前用户标识。)

当使用 **sudo** 以 **root** 用户标识执行命令时，系统将要求您输入您的口令(不是超级用户的口令)。

作为超级用户，您可以运行任何自己希望的命令*。下面举一个简单的例子，示范当您以 **root** 用户标识运行 **id** 命令时用户标识如何发生变换。不要担心您现在还不能理解全部输出。我希望您注意的是用户标识的名称(**uid**)如何变化。

```
[harley]$ id
uid=500(harley) gid=500(harley) groups=500(harley)
[harley]$ sudo id
Password:
uid=0(root) gid=0(root) groups=0(root)
```

为了方便起见，一旦您正确输入了超级用户口令，就可以在一段时间内不必再次输入口令而运行 **sudo** 命令。大多数系统的默认时间是 5 分钟，不过这一时间也可能发生变化。这意味着，如果您在 5 分钟之内使用 **sudo** 不止一次，那么您只需输入一次口令。

名称含义

su、**sudo**

su(substitute userid)命令允许您变换到另一个用户标识。**sudo**(substitute the userid and do something)命令允许您以另一个用户标识的名义执行一条单独的命令。

名称 **su** 的发音是两个单独的字母“ess-you”，而名称 **sudo** 的发音听起来像“pseudo”。实际上，刚开始听人说 **sudo** 时，您可能会迷惑，因为听起来就好像在说“pseudo”。

su 和 **sudo** 的使用是一种习惯，不仅可以作为名称，还可以作为动词。例如，我们偷

* 您可能会奇怪这不是不是一个安全问题。前面已经说过，当使用 **sudo** 命令时，系统只要求您输入自己的口令，而不是超级用户的口令。这是不是意味着，在一个共享系统上，任何人都能够使用 **sudo** 以超级用户的名义执行命令？

答案是否定的，这是因为并不是所有人都允许使用 **sudo** 命令。如果您的用户标识位于一个特殊的列表上，那么您才可以使用 **sudo** 命令。这个列表保存在文件 **/etc/sudoers** 中，而且它只能被超级用户修改(当我们在第 23 章中讨论了 Unix 文件系统之后，您就会理解该文件名称的含义)。

听一下两个喝多了酒的 Unix 人士的谈话。

人物 1: “I have an idea. Let’s edit the password file just for fun. All we have to do is pseudo the `vipw` command.(我有一个主意。让我们为了好玩编辑一下口令文件吧。我们需要做的就是使用 `sudo` 执行 `vipw` 命令。)”

人物 2: “Ahh, that’s too lame. Let’s `ess-you` instead. I like to live dangerously.(哈哈，那也太差劲了。我们还是使用 `su` 吧。我喜欢冒险的生活。)”

6.18 配置文件

我们已经讨论过，有时候为了执行特定的任务，有必要成为超级用户。如果您管理着一个拥有多个用户的系统，那么“我把口令给忘了”这类事件肯定时有发生。但是，当您是唯一用户时(在自己的 PC 上运行 Linux 或者 FreeBSD)，这种事情会经常发生吗？

答案是当您拥有自己的系统时，不必像那样经常成为超级用户。但是，在必要时，有时候也要成为超级用户。特别是，当您希望执行一个要求特殊权限的重要功能时。

大多数 Unix 程序是这样编写的，即可以通过编辑一个配置文件来定制程序。配置文件中包含有程序需要读取的信息，通常是在程序启动时读取。该信息影响程序的工作方式。

例如，在本章前面，我们讨论了当 Unix 引导到一个特定的运行级别时的启动过程。该过程依赖于一个叫 `inittab`(参见下面)的特定配置文件中的信息。

当安装新软件时，配置文件特别重要。多半情况下，软件使用一个配置文件，如果希望设置特定选项，则需要修改这个文件。在大多数情况下，随软件一起提供的文档中会详细解释配置细节。

对于某些软件来说，有一些易用的程序可以帮助修改配置文件。例如，在桌面环境中，提供有基于菜单的程序，可以使用它们来选择参数及指定选项。您可能不知道，所有这些参数和选项都存储在一个位于某个地方的配置文件中。当您“应用”修改时，程序根据您的指令将全部修改更新到配置文件中。

尽管这是一种修改配置文件的方便方法，但是更重要的是您要学习如何自己编辑这样的文件，原因有以下几点。

首先，大多数程序没有提供基于菜单的配置程序，因此，如果希望进行修改的话，您不得不自己修改。

其次，自己修改配置文件要比使用程序修改快(而且更有乐趣)。

第三，即便有菜单驱动的程序，也不可能允许访问所有可能的选项和参数。为了真正知道有什么选项和参数可用，您不得不查看配置文件。

最后，当查看配置文件时，您可以了解底层程序如何运作。程序员通常都会在配置文件中添加注释，有时候，真正理解程序更微妙部分的唯一方式就是阅读程序的注释。

提示

在 Unix 中，鼓励您查看任意系统文件的内部，包括配置文件。

在安全编辑配置文件之前，必须满足一些要求。首先，必须习惯以超级用户工作。

其次,必须能够很好地使用文本编辑器。最好的文本编辑器有 **vi**(第 22 章中讲授)和 **Emacs**。最后,一般意义上讲,必须知道自己正在做什么。稍后您将会明白这一点。

您可能奇怪,Unix 的配置文件与 Microsoft Windows 的配置文件有什么不同呢?在 Windows 中,程序将配置信息存储在两个地方:注册表或者(有时候).ini 文件。但是在 Windows 中,不鼓励 Windows 用户修改注册表。这是因为如果将注册表破坏的话,您可能会遇到大麻烦。此外,注册表内容的文档性不好。实际上,许多注册表的条目根本没有文档。

Unix 采用一种完全不同的原理。在 Unix 中没有集中的注册信息。实际上,每个程序都允许拥有自己的配置文件。此外,配置文件的内容也是文档化的。在 Unix 中,鼓励用户阅读配置文件并修改配置文件(当合适时)。

当然,如果不正确地修改了配置文件,也有可能导致问题。但是,这一问题只局限于这个特定的程序,而且在大多数情况下,这种问题也容易修复。

提示

在编辑任何重要文件(例如配置文件)之前,最好先对文件做一个备份。这样,如果出现了问题,还可以恢复原始的配置文件(第 23、24 和 25 章将介绍文件的操作)。

下面示范一下我如何去做。假设我希望编辑一个叫 **harley** 的文件,而且这天的日期是 2008 年 12 月 21 日。在开始之前,我先为这个文件复制一个副本,并将副本命名为 **harley-2008-12-21**。然后,如果出现了问题,我就可以将这个文件复制回该文件的最初名称。

为什么不使用 **harley**-或者 **harley-original** 或者 **harley-old** 这样的名称呢?这样虽然也行,但是有时候我希望在结束工作之后仍然保留备份文件。通过在文件名称中嵌入日期,我可以知道备份文件创建的时间。

6.19 浏览配置文件

在上一节中,我告诉您除非您知道自己要做什么事情,否则不要编辑配置文件。但是,没有什么理由不允许您浏览配置文件,我们可以只是看看文件里面有什么内容。

下面提供了几个有趣的配置文件。为了浏览配置文件,可以使用一个叫 **less** 的程序。**less** 的任务就是每次一屏地显示一个文件的内容。

我们将在第 21 章中详细讨论 **less**(包括它的名称)。现在要告诉您的是,为了显示一个文件的内容,只需键入 **less** 命令,后面跟着文件的名称即可。例如,为了查看名为 **/etc/passwd** 的文件,可以使用命令:

```
less /etc/passwd
```

一旦 **less** 启动,它将向您显示第一屏的信息。如果要向前移动,可以按<Space>键;向后移动,按键;查看帮助,按<h>键;退出 **less**,按<q>键。这些就是您现在所需的全部技能。

下面提供了一系列您可能感兴趣的配置文件。查看它们不需要变换成超级用户(如果要修改它们则需要)。

如果不能理解所看到的内容,不用担心。学完本书后,您就会理解了。特别是,在

阅读了第 23 章的内容之后，您就会理解这些文件名称的含义。这些文件可以在大多数 Linux 系统上找到。如果您使用一个不同类型的 Unix，则可能有一些文件拥有不同的名称。

/boot/grub/menu.lst: 关于计算机上可以引导的操作系统的信息。

/etc/hosts: 系统已知的一列主机名称和 IP 地址。

/etc/inittab: 不同运行级别的定义。

/etc/passwd: 每个用户标识的基本信息(实际口令是加密的，而且保存在其他地方)。

/etc/profile: 当一个用户标识登录时，系统自动执行的命令。

/etc/samba/smb.conf: Samba 的配置信息，Samba 是一个允许 Unix 系统和 Windows 系统共享文件和打印机的工具。

6.20 系统关闭与重新启动: init、reboot、shutdown

在本章开始时，我们讨论了登录过程中发生的事情。现在是时候讨论结束工作时应该做什么了。

基本而言，您有两个选择。一种是**关机**，这种方式将停止 Unix 并关闭计算机；另一种是**重新启动**，这种方式将先停止 Unix，然后再启动。在菜单中选择或者键入命令都可以执行这两种动作。

在桌面环境中，通过打开主菜单，选择“Logout”(或者其他相似的命令)可以关闭或者重新启动系统。作为注销过程的一部分，您可以选择关闭或者重新启动系统。如果不是这样，那么当您发现自己位于登录屏幕时，可以单击“Shutdown”或者“Reboot”关闭或重新启动系统。

使用 GUI 的方式很单调，通过输入命令关闭或者重新启动系统显然更有趣。在讲授这些命令之前，我们先回忆一下运行级别的思想。如果回头看图 6-1，那么您会发现 6 种不同的运行级别，每种运行级别将导致 Unix 以一种特殊的方式运行。

通常，为了启动系统，要在启动过程中选择一种运行级别。例如，运行级别 5 将把 Linux 启动到多用户的 GUI 模式。

但是，有两种运行级别有特殊的含义。运行级别 0 终止系统(也就是使系统关闭)，而运行级别 6 将重新启动系统。因此，当从菜单中选择“Shutdown”时，就如同修改到运行级别 0；当从菜单中选择“Reboot”时，就如同修改到运行级别 6。

您现在可能已猜到，几乎所有可以在 Unix 中通过菜单完成的事情，都可以通过键入命令完成。修改运行级别的命令是 **init**。

为了使用命令 **init**，您需要成为超级用户。一旦成为超级用户，输入该命令，后面跟上希望修改的运行级别就可以对运行级别进行修改。例如，您可以通过修改到运行级别 6 重新启动系统：

```
sudo init 6
```

如果希望关闭系统，则可以修改到运行级别 0：

```
sudo init 0
```

如果敢于尝试，您还可以修改到其他运行级别，看看会发生什么情况。

尽管可以使用 **init** 命令来重新启动或者关闭系统，但是它并不是为日常使用设计的。实际上，在日常操作中，我们通常使用其他两个命令：**reboot** 和 **shutdown**。

reboot 命令非常简单。只需输入这个命令，系统就会变换到运行级别 6：

```
sudo reboot
```

shutdown 命令稍微有点复杂，因为您必须指定何时关闭系统。选择有许多种，最简单的一种就是使用单词 **now**：

```
sudo shutdown now
```

键入这个命令就告诉系统立即变换到运行级别 0。

6.21 系统启动或者停止时发生什么事情？dmesg

在系统启动或者关闭过程中，Linux 在控制台上显示许多消息。这些消息大多数与系统硬件组件的发现和配置相关。其他消息则与启动过程中包含的服务的启动和停止相关，或者与系统一旦启动后将在后台运行的进程有关。

所有这些内容的细节已经超出了本书的讨论范围。但是，看看这些消息也很有趣。

在系统启动时，许多消息都是一闪而过，根本就来不及仔细阅读，更不用说领会它们的含义了。但是，一旦您登录系统，就可以在空闲的时候显示启动消息。只需在命令行上输入下述命令即可：

```
dmesg | less
```

尽管启动消息看上去神秘，最终您还是能够理解它们的*。

dmesg 命令的任务就是显示启动消息。但是，启动消息太多，当您只输入 **dmesg** 命令本身时，大多数消息在您阅读之前就会从屏幕上滚动过去(试一试)。

为此，可以使用上面所示由三部分组成的 **dmesg** 命令，即在 **dmesg** 命令之后跟一个 **|**(竖线)字符，后面再加上 **less**。这样将运行 **dmesg** 命令，并将它的输出发送到 **less** 命令，而 **less** 命令每次只显示一屏幕的内容。竖线创建所谓的“管道线(pipeline)”(我们将在第 15 章讨论管道线，在第 21 章中讨论 **less** 命令)。

6.22 同时做不止一件事情：II

我们从讨论人类和计算机的特征出发开始本章的讨论。在本章前面，我解释道，作为人类，我们的记忆力有缺陷，无法同时关注多件事情。我们可以做的是处理复杂的心理活动，同时每当必要时，就会在不到一秒的时间内从一种想法转换到另一种想法。

* 如果您是一个男孩，您生命中出现了一位特殊的女孩，那么下面示范一个打动她的特殊方法。

在一个星期五的晚上邀请您这位特殊女孩过来，当她过来后，对她说：“下面我给你找一个好玩的东西。”然后将 **dmesg** 命令的输出展示给她。如果知识足够渊博，那么您可以让她挑出任意消息，然后您解释给她听。根据我的研究，边听解释，边吃比萨，边喝不含咖啡因的饮料，这种方法会特别有效。

计算机在存储和恢复数据方面要比人类快速得多，而且还更准确。计算机还可以特别快地执行简单明了的任务。在处理任务时，计算机可以在不到一毫秒的时间内从一个任务转换到另一个任务。

人类可以思考和制定策略，而计算机可以分开做许多事情(同时不把各种事情混在一起)。这样，机器能够弥补我们头脑的缺点，而我们的头脑又与机器令人惊骇的能力互补。当采用正确方法时，可以使人类和计算机以这种方式融合在一起，生成一个既不是人类也不是计算机，但是比这两部分之和强大许多的结果。

将人类和计算机连在一起的胶合剂是用户界面。在 Unix 中，用户界面有两种：GUI(桌面环境)和 CLI(命令行)。

在本书的剩余部分，我们主要集中在 CLI 上。换句话说，我们将在命令行上键入命令，并学习如何使用基于文本的程序。为了做好这些，您必须熟练地同时使用终端窗口、常规窗口、虚拟终端和多桌面/工作空间。

最终的目标就是在做某些事情时，不必思考就知道如何去做。对于开车、弹奏乐器或者做饭，您应该已经知道如何去做。我希望您能够坐在计算机前面，使用 Unix 处理心理活动，执行相同类型的动作。

因为人与人各不相同，所以我不能讲授如何理解瞬间的心理活动。您需要自己开发这方面的技能。我所能做的就是通过指出您可以立即使用的神奇工具，帮助您集中注意力。首先，概括介绍一下 Linux 的默认工作环境。

在 Linux 中，有 6 个基于文本的虚拟控制台，每个控制台都拥有自己的 CLI。另外，第 7 个虚拟控制台包含有桌面环境(GUI)。

在桌面环境中，有 4 个不同的桌面/工作空间，每个都可以打开任意数量的窗口，包括终端窗口。

从一个窗口到另一个窗口，从一个虚拟控制台到另一个虚拟控制台，复制和粘贴文本都是允许的。此外，普通用户和超级用户一样，都可以进行这些操作。

在操纵工作环境时，可以使用下述快捷键。

在 KDE 中，从一个桌面切换到另一个桌面：

<Ctrl-Tab>、<Ctrl-Shift-Tab>

在 Gnome 中，从一个工作空间切换到另一个工作空间：

<Ctrl-Alt-Left>、<Ctrl-Alt-Right>、<Ctrl-Alt-Up>、<Ctrl-Alt-Down>

在桌面或者工作空间中，从一个任务切换到另一个任务：

<Alt-Tab>、<Alt-Shift-Tab>

在 GUI 中，切换到一个基于文本的虚拟控制台：

<Ctrl-Alt-F1> <Ctrl-Alt-F6>

从一个基于文本的虚拟控制台切换到另一个虚拟控制台：

<Alt-F1> <Alt-F7>(其中<Alt-F7>切换到 GUI)

在 GUI 中复制文本：

使用鼠标选择文本，然后单击鼠标中间键

在虚拟终端间复制文本：

使用鼠标选择文本，然后单击鼠标右键

在基于 GUI 的程序中，复制、剪切、粘贴数据：

<Ctrl-C>、<Ctrl-X>、<Ctrl-V>

提示

Unix 是目前拥有最好的用户界面的系统。

6.23 练习

1. 复习题

1. 什么是时间片？时间片的典型长度一般有多长？
2. 什么是 Unix CLI？什么是 GUI？
3. 什么是运行级别？此时，Fester Bestertester 正坐在演讲厅的后面听一个有关动物饲养的冗长无味的演讲。为了保持清醒，Fester 在他自己的膝上型电脑上玩一个基于 GUI 的游戏。Fester 的计算机现在的运行级别是什么呢？在某校园内，Unix 系统管理员关闭系统并重新启动系统，从而解决重要的硬件问题。这时系统管理员的计算机的运行级别又是什么呢？
4. 在您自己的 Unix 系统上，谁来充当超级用户呢？
5. 什么是虚拟控制台？如何从一个虚拟控制台切换到另一个虚拟控制台？

2. 应用题

1. 命令 **who**(在第 8 章讨论)显示当前登录到系统的用户标识列表。如果一个用户标识登录系统不止一次，那么命令 **who** 会显示出这种情况。一个接一个地登录到每台虚拟控制台，然后切换到 GUI，打开一个终端窗口，并输入命令 **who**，您会看到什么内容呢？
2. 能够从一个桌面切换到另一个桌面，而且能够快速复制和粘贴数据是十分重要的技能。在 GUI 中，打开两个终端窗口，每个终端窗口都位于自己的桌面中。在第一个桌面中，在终端窗口中输入命令 **date**(显示时间和日期)。将这个命令复制到剪贴板中。现在切换到第二个桌面，将这个命令粘贴到另一个终端窗口中。然后，使用两个基于文本的虚拟控制台重复上述练习。哪一组复制和粘贴操作比较适合您呢？为什么？

3. 思考题

1. 一些 Linux 发行版已经进行了设置，从而在系统安装过程中不创建 **root** 用户标识。在这种情况下，常规的口令充当超级用户口令。为什么 Linux 发行版要这样设置呢？这样设置有什么优点，又有什么缺点呢？
2. 为什么许多人都相信他们可以同时思考不止一件事情呢？像 Unix 这样进行快速、多任务处理的计算机系统有这样的信念会有什么结果呢？这种思考是有益的还是无益的？对您个人而言，想一想边开车边聊电话或看文本消息是不是可以？其他人这样做时又会怎么样？

Unix 键盘使用

在第 6 章中, 讨论了 GUI(图形用户界面)和 CLI(命令行界面)之间的区别。从本章开始, 在本书的剩余部分中, 将集中讨论 CLI, 这也是使用 Unix 的传统方式。

CLI 的使用方式有若干种。当使用自己的计算机时, 可以使用虚拟控制台或者终端窗口(包括 Konsole 程序)。我们已经在第 6 章中对此进行了详细讨论。当使用远程主机时, 可以通过 ssh 程序连接, ssh 程序将充当终端仿真器。无论您是如何获得 Unix 命令行的, 一旦位于命令行中, 它总是(或多或少)以相同的方式工作。

如果您使用的是基于 GUI 的系统, 那么我希望您在阅读本章内容之前, 先熟习第 6 章中的几个主题, 具体包括: 虚拟控制台、终端窗口以及如何进行选择 and 粘贴。在 GUI 中, 理解这些思想对使用 CLI 非常重要。

7.1 最初的 Unix 终端

当 Unix 刚被 Ken Thompson 和 Dennis Ritchie(参见第 2 章)开发出来时, 他们使用的终端是 Teletype ASR33 终端(参见第 3 章)。Teletype ASR33 是一种电子机械式的设备, 最初用来发送和接收文本消息。Teletype ASR33 拥有一个键盘(用于输入)和一个内置的打印机(用于输出)。Teletype ASR33 还有一个纸带穿孔机(用来在纸带上穿孔以存储数据), 以及一个纸带阅读机(用来读取穿孔纸带上的数据)。

ASR33 的能力使它适合于充当计算机终端。实际上, 从 20 世纪 60 年代中期到 70 年代中期, 几乎所有非 IBM 的计算机系统都使用 ASR33 作为控制台。Thompson 和 Ritchie 使用的 PDP 计算机也是如此, 因此, 这些设备成为最初的 Unix 终端再自然不过了(参见后面方框中的内容)。

ASR33 键盘最初是设计用来发送和接收消息的, 不是用来控制计算机操作的。因此, Thompson 和 Ritchie 不得不改编 Unix 来使用 ASR33 键盘。有趣的是他们设计的基本系统非常出色, 现在还仍然在使用。

Teletype ASR33

ASR33 由 Teletype 公司制造，于 1963 年发明。Teletype 33 有 3 种型号：RO、KSR 和 ASR。在这 3 种 Teletype 33 中，ASR 是最流行的型号。

RO(Receive-Only, 只能接收)只拥有打印机，没有提供键盘。因此，它只能接收消息，不能发送消息。

KSR(Keyboard Send-Receive, 键盘发送-接收)既提供打印机也提供键盘，可以发送和接收消息。向外发送的消息通过键盘人工键入。

ASR(Automatic Send-Receive, 自动发送-接收)拥有打印机、键盘和纸带穿孔机/阅读机。与 KSR 相似，ASR 既可以发送消息，又可以接收消息。但是，对于 ASR 而言，向外发送的消息可以通过两种方式生成。这些消息可以通过键盘手工键入，或者通过预穿孔的纸带自动读取(因此命名为“自动”)。正是由于 ASR33 集成了这些特性，才使它可以用作计算机的终端。

Teletype ASR33 终端重 56 磅，包括一个 12 磅的台子。如果您在 1974 年从 DEC 公司购买一台 ASR33，那么您需要花费 1850 美元，再加上 120 美元的安装费和 37 美元/月的维护费。按 2008 年美元价格计算，该机器的价格达到了 8400 美元，安装费有 550 美元，而维护费每月有 170 美元。

第 3 章中给出了 ASR33 的照片。图 3-1 和图 3-2 展示了该机器。图 3-3 则是纸带穿孔机/阅读机的特写镜头。

正如您所期望的，Teletype 的键盘包含有字母表中的 26 个字母、数字 0-9 以及最常见的标点符号。但是，键盘上还有几个特殊的键，用来提供发送和接收消息所需的功能(参见图 7-1)。这类键中最重要的键有：<Esc>、<Ctrl>、<Shift>、<Tab>和<Return>。

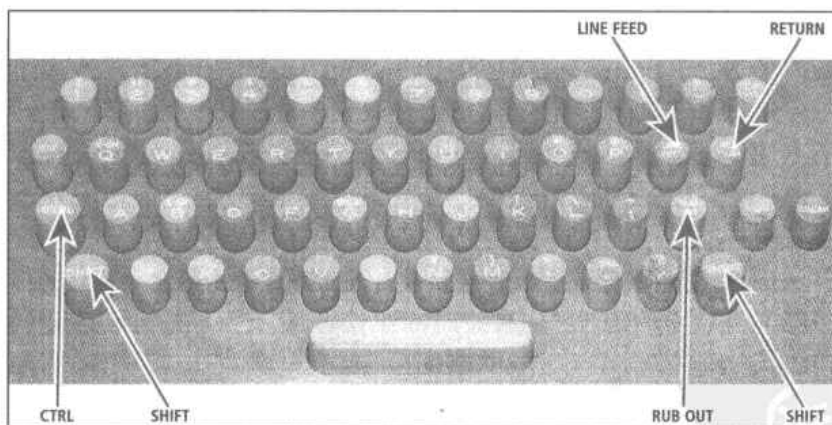


图 7-1 Teletype ASR33 的键盘

对实际应用来说，Teletype ASR33 键盘上最有趣的键如下所示。从顶部起第二排，最右边的是 RETURN 键。RETURN 键左边的键是 LINE FEED。从底部起第二排，右边第三个键是 RUB OUT。CTRL 键位于最左边。在最底部那一排，两个最边上的键都是 SHIFT。注意该键盘上没有 BACKSPACE 键。

<Ctrl>(Control, 控制)键非常有用, 因为它像<Shift>键一样, 可以与其他键组合起来形成新的组合键。例如, 按下<Ctrl>键时按下下一个字母键或者数字键, 将发送一个诸如<Ctrl-A>、<Ctrl-B>、<Ctrl-C>之类的信号。

Thompson 和 Ritchie 所做的就是将这些键的使用集成到操作系统的基本设计中。为了做到这一点, 他们编写的 Unix, 可以使用特定的信号控制正在运行的程序的操作。例如, 称为 **intr**(interrupt, 中断)的信号用来终止程序, 而按下<Ctrl-C>可以发送 **intr** 信号。

用技术术语讲, 当两件事情之间存在等价关系时, 就可以说它们之间存在映射(mapping)。当我们创建这样的等价关系时, 就可以说我们将一件事情映射到另一件事情。例如, 如果我们说 A 映射到 B 上, 那么这意味着, 当我们使用 A 时, 就如同使用 B 一样。

映射是一个非常重要的概念, 一个您在使用计算机时会时常遇到的概念。对于上述例子, 我们可以说, 在 Unix 中, <Ctrl-C>字符映射到 **intr** 信号上。这也就是说, 当按下<Ctrl-C>时, 它的效果就是发送 **intr** 信号。

一会儿, 我们将详细讨论 Unix 信号。实际上, 本章的主要目标是解释重要的信号和它们的键盘映射。但是, 我希望首先花点时间讨论一下术语。

7.2 Teletype 和 Unix 文化

在使用 Unix 时, 您将会发现许多约定基于 20 世纪 70 年代(即早期版本的 Unix 开发期间)的技术。特别是, Unix 世界中许多思想基于早期终端的特点。这就是为什么在本章和第 3 章中要花时间讨论 Teletype(最初的终端)和 VT100(最流行的终端)的原因。

在本节中, 我希望快速地介绍两个今后会大量遇到的约定。

我希望您注意的第一个约定是缩写“tty”(发音为“tee-tee-why”)。许多年以前, Teletype 机器仍然在使用的时候, 它们被称为 TTY。这种习惯被 Unix 采纳。即便使用 Teletype 的时代已经过去了很久, 人们仍然经常使用单词“tty”作为 Unix 终端的符号。尤其是在 Unix 文档以及程序和命令的名称中, 您将会看到这个术语大量存在。下面举一些例子:

- 在 Unix 系统中, 每个终端都有自己的名称。显示自己终端名称的命令是 **tty**(大家可以试一试, 看看得到什么结果)。
- 命令 **stty**(“set tty”, 设置 tty)用来显示或改变终端的设置。
- 程序 **getty**(“get tty”, 获取 tty)用来打开与一个终端的通信, 并启动登录进程。

我希望提及的第二个约定与打印有关。Teletype 终端有两种输出数据的方法。它们可以在连续的 8.5 英寸卷筒纸上打印适合人类阅读的数据*, 或者在 1 英寸宽的纸带上穿孔以得到适合机器阅读的数据。如果查看第 3 章中的图 3-2, 就会发现卷筒打印机纸(位于中间)和纸带卷筒(位于左边)。

因为输出就意味着打印, 所以在 Unix 中习惯使用单词打印(print)描述信息的输出。在那个时候, 这是合理的, 因为从字面上讲, 输出就是在纸上打印。但是, 有趣的是, 即便

* 为了满足您的好奇心, 下面详细介绍一下 Teletype 的打印。Teletype 在连续的 8.5 英寸卷筒纸上打印输出, 卷筒的直径大约有 5 英寸。Teletype 机器输出时, 每英寸输出 10 个字符, 每行最多有 72 个字符。垂直间距是每英寸 6 行。Teletype 机器的打印是单色的, 通常是黑色。

是出现了更加现代的终端，并且数据可以显示在显示器上以后，人们仍然使用单词“print”，而且时至今日，情况依然如此。

因此，在 Unix 文档中，无论何时，当您阅读到“打印数据”时，差不多总是指数据的显示。例如，前面提到的 **tty** 命令是显示终端的内部名称。如果您在 Linux 版本的 Unix 联机手册(参见第 9 章)中查看该命令的话，就会发现 **tty** 的目的是“print the file name of the terminal connected to standard input(显示与标准输入相连的终端的文件名称)”。

下面再举一个例子。在使用 Unix 文件系统时，当前所在的目录称为“工作目录”(我们将在第 23 章中讨论这些思想)。显示工作目录名称的命令是 **pwd**，它的含义为“print working directory(显示工作目录)”。

此时，您可能不禁要问：如果“print(打印)”意味着“display(显示)”的话，那么当真正指打印时该使用哪个术语呢？

这个问题有两个答案。第一，在一些情况下，“print(打印)”用于指真正的打印，从上下文可以清楚地明白它的含义。

其他时候，您将看到术语“line printer(行打印机)”(它本身也是一个过时的术语)或者其缩写“lp”。当看到这个时，您就可以认为它代表“printer(打印机)”。例如，两个打印文件的最重要命令分别为 **lp** 和 **lpr**(**lp** 起源于 System V，**lpr** 起源于伯克利的 Unix)。

7.3 Termcap、Terminfo 与 curses

正如第 3 章中解释的，Unix 被设计成人们使用终端访问主机计算机的系统。Unix 开发人员需要解决的一个最重要的问题就是每种类型的终端都拥有自己的特征，并且使用自己的命令集。例如，尽管所有的显示终端都有一个清除屏幕的命令，但是所有终端中这一命令的名称可能不尽相同。

因此，如果您在编写程序，并且在某个特定时刻，需要清除用户终端的屏幕，该怎么办呢？当实际命令依赖于使用的是哪种类型的终端时，您怎么知道该使用哪个命令呢？

要求每个程序都知道每种类型终端的每个命令并不合理。这将是软件开发人员的一个极大的负担*。此外，当又有新型终端开发出来后，又该怎么办呢？已有程序能否正常使用新型终端呢？

解决方法就是将所有不同类型的终端的描述收集到一个数据库中。然后，当程序希望向终端发送命令时，它可以通过使用数据库中的信息以一种标准化的方式完成(稍后我们还会详细讨论它如何进行)。

第一个这样的系统由 Bill Joy 创建，Bill Joy 是伯克利 Unix 之父(参见第 2 章)。1977 年，Joy 还是一名研究生，他在封装 1BSD(伯克利 Unix 的第一个官方版本)的过程中，包含了一个管理各种类型终端的显示屏幕的系统。1978 年中，他发布了 2BSD，对该系统进行了完善，并将这个系统命名为 Termcap(terminal capabilities)。第一个使用 Termcap 的重要程序是 **vi** 编辑器(参见第 22 章)，该程序也是由 Joy 编写的。

* 即便是在 1980 年，大多数终端所支持的命令也超过了 100 个。

要在一个程序中使用 Termcap 需要很多的工作。为了使它方便简单些, 另一个伯克利的学生 Ken Arnold 开发了一个程序界面, 即 **curses**(该名称指的是 cursor addressing)。**curses** 用来执行屏幕显示管理所需的所有功能, 同时对程序员隐藏细节。

一旦程序员学会了如何使用 **curses**, 那么他就可以编写使用任何类型终端的程序, 即便是还没有发明的终端。所需的全部条件就是终端在 Termcap 数据库中有一个入口。第一个使用 Termcap 的程序是一个非常流行的基于文本的游戏, 即 **Rogue**(参见下面方框中的内容)。

curses 和 Termcap 的第一次使用: 游戏 Rogue

第一个使用 **curses** 和 Termcap 来控制显示屏幕的程序是 **Rogue**, 一个单玩家、基于文本的城堡探险类型的魔幻游戏。

在玩 **Rogue** 时, 您要承担在一个巨大的城堡中探险的角色。游戏刚开始时, 您位于城堡的顶层。您的目标就是打开一条通往城堡底层的路, 并在城堡底层拿到 Yendor 的护身符。然后您必须携带护身符返回到城堡顶层。在前进道路上, 到处都充满着怪物、陷阱、秘道和财宝。

在 **Rogue** 开发出来时, 还有一个单玩家的魔幻游戏 **Adventure**。这是一个在程序员中间非常流行的游戏(实际上, 我还记着在古老的 Texas Instruments 显示终端上玩这个游戏的情形, 那时终端通过低速的电话线连接到 Unix 计算机上)。在 **Adventure** 游戏中, 每次玩时, 探险过程都是相同的, 但是在 **Rogue** 中, 城堡及其内容都是随机生成的。这意味着游戏永远是不同的。另外, 因为 **Rogue** 使用了 **curses**, 所以它能够绘制简单的地图, 这一点是 **Adventure** 无法做到的。

Rogue 的作者是 Michael Toy、Glenn Wichman, 后来又加入了 Ken Arnold。该游戏的第一版是为伯克利 Unix 编写的, 1980 年, **Rogue** 被包含到 4.2BSD 中。4.2BSD 非常流行, 在极短的时间内, 全世界的学生就开始玩 **Rogue**。

如果能够看到第一版的 **Rogue**, 您就会发现它相当原始。但是, 它比以前的任何计算机游戏都要复杂, 而且在那个时候, 人们认为它特别酷。最终, **Rogue** 被移植到其他许多系统上, 包括 PC、Macintosh、Amiga 和 Atari ST。

现在, **Rogue** 游戏仍然存在, 全世界的人们都还在玩它的现代版本。如果您希望看一看 Unix 早期时代最有趣的遗产之一, 可以在 Internet 上搜索 “**Rogue**”。

为了有效地工作, Termcap 数据库必须包含每台 Unix 可能使用的终端的每种变体, 而且所有这些数据都必须包含在一个单独的文件中。多年以来, 随着新型终端的不断出现, Termcap 文件变得日益庞大, 从而使它难以维护, 搜索变慢。

此时, 为了开发 System III 以及后来的 System V Release 1(参见第 2 章), 贝尔实验室的程序员增强了 **curses**。为了提高 **curses** 的性能, 贝尔实验室的程序员用一个新的工具 **Terminfo**(即 terminal information)替换了 Termcap。Terminfo 将数据存储在一组文件中, 每种终端类型一个文件。这些文件组织到 26 个命名为 **a** 至 **z** 的目录中, 它们都保存在 Terminfo 目录下(在您阅读完第 23 章的内容后就会理解)。Terminfo 的设计非常灵活, 因此现在仍在使用。例如, 在 Linux 中, 通用 VT100 终端的信息就存储在下述文件中:


```
/usr/share/terminfo/v/vt100
```

主 Terminfo 目录的位置根据系统的不同可能有所不同。为了便于您在自己的系统中查找该位置，下面列举一些最常使用的名称：

```
/usr/share/terminfo/  
/usr/lib/terminfo/  
/usr/share/lib/terminfo/  
/usr/share/misc/terminfo
```

Terminfo 的最大问题在于 AT&T 公司，它拥有贝尔实验室，而且它不同意发布源代码（参见第 2 章）。这意味着尽管 System V 拥有 Terminfo 和一个更好的 **curses** 版本，但是黑客社区不能访问它。他们不得不凑合着使用古老的、功能欠强大的基于 Termcap 的工具。

为了克服这一限制，1982 年，一个名叫 Pavel Curtis 的程序员开始着手开发一个自由版本的 **curses**，他称之为 **ncurses**（即 new curses）。直到另一名程序员 Zeyd Ben-Halim 于 1991 年接手该项目时，**ncurses** 都只有非常有限的发行。1993 年末，Eric Raymond 加入到 Ben-Halim 行列中，他们俩一起坚定不移地着手 **ncurses** 的开发。

在 20 世纪 90 年代初期，**ncurses** 面临着许多问题。但是，随着其他人的加入，他们一起努力及时解决了这些问题，最终 **ncurses** 和 Terminfo 一起合并成一个不朽的标准。

现在，Terminfo 已经永久取代了 Termcap。但是，为了保持与非常古老的程序的兼容，一些 Unix 系统仍然拥有 Termcap 文件，即使 Termcap 已经过时并且成为不推荐产品*。

您希望看一看 Termcap 或者 Terminfo 信息是什么样子的吗？Termcap 数据库容易显示，因为它包含的是纯文本，存储在一个长文件中。如果系统中有 Termcap 文件，那么可以使用下述命令显示它：

```
less /etc/termcap
```

less 程序显示文件的内容，每次一屏。我们将在第 21 章中详细地讨论 **less** 程序。现在，我告诉您一些 **less** 命令启动之后的操作：

- 向前移动一屏，按<Space>键。
- 向后移动一屏，按键。
- 退出，按<Q>键。
- 显示帮助，按<H>键。
- 跳到 VT100 在 Termcap 中的条目，键入/^vt100 并按<Return>

其中，字符/（斜线）意味着“search(搜索)”，而字符^（插入记号）意味着“at the beginning of a line(在某行的开头)”。

Terminfo 数据已经进行了编译（也就是说处理成非文本格式），这意味着不能直接看到它的内容。因此，您必须使用一个特殊的程序（这个程序叫 **infocmp**）来读取数据并翻译成纯文本（如果系统不识别 **infocmp** 命令的话，可能意味着系统中没有安装 **ncurses**）。

* 如果某些东西已经被列为不推荐产品，那么这意味着，尽管可以继续使用它，但是最好不要使用，因为它已经过时。

您可能经常在计算机文档中看到术语“deprecated(不推荐)”，特别是在编程领域，在这一领域事情变化很快。当您看到这样一个注释时，应该注意被注释特性很有可能在产品的未来版本中消失。

在使用 **infocmp** 时, 只需指定希望查看其信息的终端名称即可, 例如, 显示 VT100 终端的 Termino 数据, 可以使用下述命令:

```
infocmp vt100 | less
```

字符|(竖线)将 **infocmp** 命令的输出发送给 **less**, 从而可以一次一屏地显示输出。

如果要显示当前正在使用的终端的 Termino 数据, 可以使用不提供终端名称的 **infocmp** 命令:

```
infocmp | less
```

如果是在 PC 上运行 Linux, 那么看到的内容有两种可能性。从终端窗口, 例如 Konsole 程序(参见第 6 章)来看, 将得到一个 **xterm**(一种 X 终端)类型的终端。从虚拟控制台(参见第 6 章)来看, 将得到一个 **linux** 类型的终端, 它有点像 VT220(VT100 的彩色版)。

7.4 Unix 如何知道所使用终端的类型

从前面的讨论中可以看出, Unix 需要知道正在使用的终端的类型, 这一点非常重要。在 20 世纪 90 年代末之前, 这还是一个问題。那个时候, 终端有许多不同的类型, 需要用户准确告诉系统正在使用的终端的类型。这样做要求用户学习使用各种各样的技术命令。

现在, 这已经不再需要, 原因有两方面。首先, 许多人在自己的 PC 机上使用 Unix。这时, “终端”是内置在计算机中的, Unix 当然知道使用的是什类型终端。

当连接到远程主机(通过局域网、Internet 或者电话线)时, 使用的是终端仿真程序, 并不是真正的终端。在这种情况下, 这些程序能够告诉远程主机它们正在仿真什类型的终端, 所以您不必考虑细节问题。

尽管使用程序有可能仿真任何类型的终端, 但是在实际中, 现在只能看到 4 种类型的终端。其中两种最常见的终端是 VT100 和 **xterm**。VT100 是众所周知的基于文本的终端, 我们已经在第 3 章中详细地讨论了它。**xterm** 是一个 X 终端, 即第 3 章中讨论的标准图形终端。下面稍微扩展一点, 介绍另外两种仿真的终端: VT220(VT100 的彩色版)和 3270(IBM 公司的大型机使用)。如果使用的是 Linux 系统, 则还可以看到另外一种终端类型 “linux”, 它实质上就是 VT220。

为了记录正在使用的终端的类型, Unix 使用了环境变量。环境变量是一个拥有名称和值的条目, 通常由 shell 和所有运行的程序使用。另外还有一个全局环境变量 **TERM**, 它的值设置为您所使用终端的类型。

在任何时间, 使用命令 **echo** 后面跟一个\$(美元符号)字符和变量的名称都可以显示任何环境变量的值。例如, 查看 **TERM** 变量的值的命令如下所示:

```
echo $TERM
```

该命令将显示当前正在使用的终端的类型。

我们将在第 12 章中更详细地讨论环境变量。现在, 如果您比较好奇, 那么下面给出一个命令, 该命令将显示您的 shell 中所有环境变量的值。

printenv

这个命令的名称 **printenv** 代表 “print environment variables, 显示环境变量”。您应该还记着, 在本章前面, 我解释过 Unix 使用单词 “print(打印)” 作为 “display(显示)” 的同义词。

提示

在第4章中, 解释过 Unix 系统区分大小写字母。例如, 在 Unix 中, 名称 **harley** 与名称 **Harley** 完全不同。

因为小写字母容易键入, 所以习惯于使用小写字母作为专有名称, 包括用户标识、命令和文件。这就是为什么可以看到用户标识 **harley**, 但是永远看不到 **Harley** 或者 **HARLEY** 的原因。

这一规则的主要例外与环境变量相关。在 shell 中, 环境变量使用大写字母命名, 例如 **TERM**, 这是传统惯例。通过采用这种方式, 一瞥之下, 就可以知道它是环境变量。

7.5 修饰键: <Ctrl>键

<Ctrl>键(名称代表 Control, 即控制)是本章开头提到的早期 Teletype 终端的一个特性。在图 7-1 所示的 Teletype 键盘中, <Ctrl>键是从底部起第二排最左边的键。顾名思义, <Ctrl>键用于控制 Teletype 的操作。当创建 Unix 时, Unix 开发人员采纳了<Ctrl>键, 并将它以若干种方式集成到系统中, 稍后我们将对此展开讨论。

在使用<Ctrl>键时, 需要将它按住(就像<Shift>键一样), 然后按另一个键, 通常是一个字母键。例如, 您可以按住<Ctrl>键的同时按<A>键。

根据字母表, 这种组合有 26 种, 即从<Ctrl>+<A>到<Ctrl>+<Z>, 其中有几个大家可能会遇到。因为一遍又一遍地写 “Ctrl” 不方便, 所以 Unix 社区使用一种简写表示方法: 字符^ (插入记号)。当看到这个字符后面有另一个字符时, 它意味着 “按住<Ctrl>键”。例如, ^A 意味着按住<Ctrl>键并按下<A>键。另外还有<Ctrl-A>或 **C-a** 等表示方法, 它们的含义都相同。

根据约定, 在写<Ctrl>组合键时通常使用大写字母。例如, 我们通常写 ^A, 从来不写 ^a。使用大写字母使这样的组合键容易阅读, 对此可以比较 ^L 和 ^l。但是, 它并不是一个真正的大写字母, 因此, 当使用<Ctrl>组合键时, 并不需要按下<Shift>键。

为了习惯这种表示方法, 我们先看一看下面的例子。这是本章稍后要提到的 **stty** 命令的一部分输出。

```
erase kill werase rprnt flush lnext
^H   ^U   ^W   ^R   ^O   ^V

susp  intr  quit  stop  eof
^Z/^Y ^C   ^\   ^S/^Q ^D
```

stty 命令的输出告诉我们按下哪些键组合会发送特定的信号。细节现在并不重要, 我

希望您注意的是表示方法。在这个例子中,发送 **erase** 信号要使用 **^H**。也就是说,按住 **<Ctrl>** 键并按 **<H>** 键。对于 **kill** 信号,要使用 **^U**; 而对于 **werase**, 则使用 **^W**; 等等。

<Ctrl> 键是所谓的修饰键的一个例子。修饰键指按住这些键的时候再去按另一个键。例如,当键入 **^H** 时, **<Ctrl>** 键“修饰” **<H>** 键。

在标准的 PC 键盘上,修饰键有 **<Shift>**、**<Ctrl>** 和 **<Alt>**。**<Shift>** 键有两种用途: 键入大写字母及键入双字符键中的顶部字符。例如,键入美国标准键盘上的 **&**(和号)字符,需要按 **<Shift-7>**。**<Ctrl>** 键用于键入特殊信号,其方式前面已经说明过。**<Alt>** 键是最新的修饰键。因为当 Unix 开发出来时,该键还不存在,所以它不是标准 Unix 键盘的一部分。因此,在使用标准 Unix CLI(命令行界面)时不需要该键。但是,正如第 6 章中讨论的,GUI 使用 **<Alt>** 键。

7.6 Unix 键盘信号

最初的 Unix 设计假定人们使用终端连接主机计算机。30 多年过去后,情况依然如此,即便是在自己的 PC 机上运行 Unix^{*}。多年以来,终端发展为许多不同的类型,并且提供了许多不同类型的键盘,但是 Unix 一直能够很好地使用它们。这是因为 Unix 使用了一个键盘映射系统,该系统非常灵活,可以用于任意类型的键盘。

为了控制程序运行时的操作,Unix 使用了一组键盘信号。尽管这些信号是标准的,但是发送这些信号所需按下的键却可以根据需要变化。正是这一点创造了灵活性。例如,有一个叫 **intr**(interrupt, 中断)的信号可以终止进程的运行。如果输入了一条需要花费很长时间才能结束的命令,那么就可以向该命令发送 **intr** 信号来停止该命令的运行。

intr 信号的概念已经嵌入到 Unix 的定义中。没有嵌入到 Unix 中的是发送该信号所需按下的实际按键。理论上,发送该信号时可以使用任意的有效键或者键组合,而且终端与终端之间还可以不同。

对于大多数终端来说, **^C**(**Ctrl-C**)键映射到 **intr** 信号上。换句话说,按下 **^C** 键就可以停止程序。但是,也有少数几个终端将 **<Delete>** 键映射到 **intr** 信号上。对于这些终端,按下 **<Delete>** 键就可以终止程序。不管是哪一种情况,如果您不喜欢这种键盘映射,就可以修改它。

通常,您并不希望修改 Unix 键盘映射,但是您可以,而且正是这一点才使系统的适应性如此之强。在下面几节中,将描述一些重要的键盘信号,并说明如何使用它们。然后将示范如何查找特定终端所使用的键,以及在希望时如何修改键盘的映射。

理解键盘信号以及如何使用它们是使用 Unix CLI 所需掌握的基本技能之一。

7.7 键入过程中使用的信号: **erase**、**werase**、**kill**

在键入过程中有 3 个键盘信号可以使用: **erase**、**werase** 和 **kill**。**erase** 删除最后一个

^{*} 正如第 6 章讨论的,当在自己的计算机上运行 Unix 时,每个虚拟控制台和每个终端窗口都是一个单独的终端,它们都连接到同一个主机计算机上。

键入的字符，**werase** 删除最后一个键入的单词，而 **kill** 则删除整行。

按下<Backspace>或者<Delete>键(取决于键盘及其映射)通常可以发送 **erase** 信号。看一看键盘主部分右上角的大键，在几乎所有情况下，这就是映射到 **erase** 信号的键。在键入信息时，按下这个键可以删除刚刚键入的最后一个字符。

对于大多数终端和 PC 来说，使用的是<Backspace>键。而对于 Macintosh 来说，使用的是<Delete>键。如果使用的是 Sun 公司的计算机，它的键盘拥有这两个键，且两者紧挨在一起，那么这时使用<Delete>键(靠上的键)。

提示

对于大多数键盘，按下<Backspace>键发送 **erase** 信号。而对于 Macintosh 来说，按下<Delete>键发送 **erase** 信号。您使用哪个键取决于您自己的键盘。

您应该还记着，在第4章中提到一些键盘有<Enter>键，而其他键盘有<Return>键。道理是相同的。Unix 只关心当按下这个键时发送的是什么信号。只要按下正确的键，Unix 不会关心它是如何命名的。

在本书中，当提及<Backspace>键时，或者指<Backspace>，或者指<Delete>，哪一个都可以在系统上使用，只要您的键盘提供。同理，当提及<Return>键时，或者指<Return>，或者指<Enter>，选择使用哪一个取决于您的键盘。

下面示范一个使用 **erase** 信号的例子。假设希望输入命令 **date**(显示时间和日期)，但是将它拼错成 **datx**。在按<Return>键之前，按<Backspace>(或者<Delete>)键删除最后一个字母，然后再进行纠正：

```
datx<Backspace>e
```

在计算机屏幕上，当按下<Backspace>键时 **x** 将消失。如果希望删除不止一个字符，可以连续按<Backspace>键。

下一个信号 **werase**，告诉 Unix 删除刚刚键入的最后一个单词。**werase** 信号对应的键通常是 **^W**。当希望纠正一个或者多个刚刚键入的单词时，这个键非常有用。当然，也可以重复地按<Backspace>键，但是当希望删除整个单词时，**^W** 要快许多。

下面举一个例子。您是一名间谍，希望使用 **less** 程序显示 3 个命名为 **data**、**secret** 和 **top-secret** 的文件的内容。使用的命令为：

```
less data secret top-secret
```

您键入了命令，但是，在按下<Return>键之前，您注意到另一名间谍站在您身后，从您的肩膀上偷看过来。因此，您决定最好不再显示 **secret** 和 **top-secret** 文件。所以您按下 **^W** 两次，删除最后两个单词：

```
less data secret top-secret^W^W
```

在计算机屏幕上，首先是单词 **top-secret** 消失，然后是单词 **secret** 消失。接下来您可以按下<Return>键运行该命令。

在键入时使用的第三个信号是 **kill**。映射到 **kill** 信号的键通常是 **^X** 或者 **^U**(取决于系

统如何设置)。该信号告诉 Unix 删除整行。

例如，假如您准备显示前面提到的 3 个文件的内容。您键入了命令，但是在按<Return>键之前，有人跑进房间，告诉您他们正在银行里免费发钱。快速地思考之后，您按下^X(或者^U)键删除整行命令：

```
less data secret top-secret^X
```

在计算机屏幕上，整行内容消失。您现在可以注销系统，跑去银行(当然，永远不要在登录的情况下离开终端，即便是跑出去拿免费的钱)。

出于参考目的，图 7-2 中总结了键入时使用的键盘信号。

信号	键	作用
erase	<Backspace>/<Delete>	删除最后一个键入的字符
werase	^W	删除最后一个键入的单词
kill	^X/^U	删除整行

图 7-2 键入时使用的键盘信号

提示

kill 键盘信号不会停止程序。它只删除刚键入的一行。为了停止程序，需要使用 intr 信号，该信号对应的键是^C 或者<Delete>。

7.8 <Backspace>和<Delete>

正如前面所述，Unix 的设计使用了早期 Teletype 终端上可用的基本键：字母表中的字母、数字、标点符号、<Shift>键、<Return>键和<Ctrl>键。实际上，时至今日，您仍然可以只依靠这些基本键使用 Unix CLI。

现代的键盘添加了其他键，例如<Backspace>、<Alt>、<Pageup>、<Pagedown>、功能键、光标控制(箭头)键等。这些键中最有趣的要属<Backspace>键(在 Macintosh 机上为<Delete>键)。为了理解这个键为什么如此有趣，我们需要回到过去，访问我们的老朋友 Teletype ASR33，即最初的 Unix 终端。以这种方式，就可以解释下一节将出现的神秘字符^H。

大家应该还记得，Teletype 不止拥有键盘和打印机，它还拥有纸带穿孔机和纸带阅读机。穿孔机用来在纸带上穿孔，而阅读机用来读取纸带上的孔，并将它们翻译成数据。

纸带是一英寸宽，1000 英尺长的卷纸。纸带上的每一行通过在 8 个位置上穿孔可以存储 1 字节(8 位)的数据。纸带每英寸可以存储 10 字节的数据。当读取纸带时，孔的存在或者不存在被解释成一个二进制数字：有孔表示 1；没有孔表示 0(如果您不理解二进制算术，现在还不用担心)。

现在您或许会问，为什么这对于现代的 Unix 系统非常重要呢？原因是：1970 年纸带的物理配置对现在计算机上<Backspace>键的工作方式有直接的影响。下面详细阐述其原理。

假设您正在使用一台 Teletype 机器，而且正在键入穿孔到纸带上的信息。每键入一个字符，机器就在纸带上穿一组孔。这些孔对应于键入字符的 ASCII 码(参见第 19 章)。您的工作一直很好，直到您偶然犯了一个错误。现在您该怎么办呢？

对于现代的 PC 机来说，只需简单地按下<Backspace>键即可。然后显示器上的光标向前移动一个位置，刚刚键入的字符从屏幕上消失。但是，当在纸带上穿错孔时，处理就没有如此简单了。

解决方法分为两部分，并且涉及特殊的 Teletype 命令。首先，按下<Ctrl-H>键，向纸带穿孔机发送一个 BS(backspace, 退格)命令。这致使穿孔机移回到前一行(发生错误的位置)。然后，按下<Rubout(擦掉)>键，发送 DEL 命令。这将致使穿孔机在 8 个位置上都穿上孔。

这样处理的原因在于纸带阅读机被设置为跳过任何 8 个位置都穿孔的字符。在基 2 的语言中，我们可以说纸带阅读机忽略任何都是由 1 构成的二进制模式。

因此，通过将一个字符附加穿 8 个孔(也就是说，通过将字符的二进制码转换成 8 个 1)，可以有效地删除该字符。这就是为什么完成这一任务的键命名为“Rubout”的原因(您可以想象成用铅笔写字时犯了错误，然后用橡皮将错误擦掉)。

因此，对于早期的 Unix 开发人员，删除一个错误需要两步不同的操作：使用^H 退格，然后使用<Rubout>将错误擦掉。他们面临的问题是，哪一个键应该映射到 erase 信号，^H 还是<Rubout>呢？他们选择了^H。

很快，计算机公司开始制造有<Backspace>键的终端。为了方便，这个键被编程为具有与按下^H 相同的含义。当您犯了键入错误时，可以通过按下<Backspace>或者^H(它们都映射到 erase 信号上)删除错误。

后来，<Rubout>键的名称被改变为<Delete>，这样才合理。最终，一些 Unix 公司(特别是 Sun 公司)决定在它们的键盘中添加一个实际的<Delete>键。与旧的<Rubout>键相似，<Delete>键也生成 DEL 码(最初用于“删除”纸带上的一个字符)。然后这些公司决定使用 DEL 码取代^H 代表退格。

因此，现实变成：对于一些键盘来说，可以按下<Backspace>键删除一个字符，而对于其他键盘来说，要按下<Delete>键。在第一种情况中，<Backspace>等同于 Teletype 上的^H，用来发送 BS 码。在第二种情况中，<Delete>等同于 Teletype 上的<Rubout>，用来发送 DEL 码。

使事情更加混乱的是，当编写 Unix 文档时存在一个问题。BS 码有一种简单的表示方法，即^H，但是 DEL 码没有一种简单的表示方法。为了解决这个问题，人们选择了^?标记来表示 DEL。

但是，与^H 不同，^?并不是一个真正的键组合。也就是说，您不能按住<Ctrl>键再按?(问号)键。^?只是两个字符的缩写，意味着“whichever key on your keyboard that sends the code that used to be called DEL(您的键盘上某一个用来发送 DEL 代码的键)”。

但是，使事情更糟糕的是，后来 Unix 系统被配置成<Backspace>等同于^?(DEL)，而不是^H(BS)。在这种情况下，^?映射到 erase，而不是^H。

情况就是这样。如果您的键盘上有<Backspace>键，那么它将映射到 erase 信号。PC 机是这种情况。如果没有<Backspace>键，那么您的键盘上将有一个<Delete>键映射到 erase

信号。Macintosh 机是这种情况。

如果您使用的是 Sun 公司的计算机，那么键盘上既有<Backspace>键又有<Delete>键，这时<Backspace>键将等同于^H，而<Delete>键将等同于^?。这两个键中有一个映射到 **erase** 信号。大家可以试一试，看看是哪一个键。

为了避免^H 和^?之间的混淆，一些 Unix 系统定义了一个额外的信号 **erase2**。例如，FreeBSD 就是这样的情况。

erase2 的效果与 **erase** 相同。也就是说，它删除刚键入的最后一个字符。它们之间的区别是^H 映射到一个信号——**erase** 或者 **erase2**，而^?映射到另一个信号。这样，无论<Backspace>发送^H 还是^?，都能正常工作。

提示

1981 年 8 月，当 IBM PC 刚生产出来时，它提供了一种全新的键盘。该键盘(与现在我们使用的键盘几乎相同)拥有好几个新键，例如<Insert>、<Delete>、<Pageup>、<Pagedown>等等。

重要的是要意识到 PC 机键盘上的<Delete>键与旧终端、Macintosh 机或 Sun 公司的计算机上的<Delete>键并不相同。它是一个完全不同的键。

如果拥有一台 PC 机，那么您可以自己证实这一点。在 Unix 命令行中输入一些内容，但是不要按下<Return>键。现在按<Backspace>键几次。注意最后几个字符被删除了。这是因为，在您的计算机上，<Backspace>键或者等同于^H，或者等同于^?，无论何种情况，<Backspace>键都会映射到 **erase** 信号上。

现在按<Delete>键(<Insert>键的旁边)。它不删除前面的字符，因为它没有映射到 **erase** 信号上。

7.9 神秘字符^H

假设您正在使用自己的 PC 机通过网络连接到一台远程 Unix 主机。您正在键入命令，突然，有人朝您的头上扔了一个红白色的 Betty Boop 玩偶。这使您受到了惊吓，当您再看屏幕时，发现犯了一个键入错误。

您连续按<Backspace>键几次，却发现没有删除最后几个键入的字符，而是在屏幕上显示：

```
^H^H^H
```

您吃惊地看着屏幕。为什么<Backspace>键会显示^H，而不是删除字符呢？难道是^H 没有映射到 **erase** 信号上吗？

答案是在您的计算机上<Backspace>键等同于^H，而^H 被映射到 **erase** 信号上。这就是为什么在您的机器上，<Backspace>键工作正常的原因了。

但是在远程主机上，^?映射到 **erase** 信号上了。当您按下<Backspace>键时，发送的^H 在远程主机上没有意义。这就是在远程主机上<Backspace>没有发挥作用的原因。

您有 4 个选择。首先，您可以使用其他键修复键入错误。即不再使用<Backspace>键每次删除一个字符，而是使用[^]W 删除整个单词，或者使用[^]X/[^]U 删除整行。

其次，您可以找出向远程主机发送[^]?的键。在大多数系统上，<Ctrl-Backspace>可以完成这一任务。试试它，看看这个组合键能不能正常删除字符(如果使用 Macintosh 机，可以尝试一下<Option-Backspace>)。

第三，您可以修改连接远程主机的程序的配置。大多数通信程序允许控制<Backspace>键发送[^]H 还是发送[^]?。如果程序允许这样做，那么您可以配置该程序，从而保证无论何时，当您连接这种主机时，<Backspace>发送的是[^]?，而不是[^]H。

最后，您可以改变远程主机的映射，从而将[^]H 而不是[^]?映射到 **erase** 信号上。一旦这样做了，<Backspace>将正常工作。这通常是最佳的解决办法，特别是在需要大量使用远程主机的情况下。

在进行该修改时，您所需做的全部工作就是在每次登录到远程主机时都要执行的初始化文件中放置一条特定的命令。命令如下：

```
stty erase ^H
```

我们将在本章后面详细讨论 **stty** 命令的用法。

根据所使用 shell 的不同，放置该命令的文件的名称也有所不同。如果您使用的是 Bash(Linux 系统的默认 shell)或者 Korn Shell，那么需要将该命令放置在您的 **.profile** 文件中。如果使用的是 C-Shell，那么需要将该命令放置在 **.login** 文件中。在两种情况中，“.”(点号)都是文件名的一部分(我们将在第 14 章中讨论这些文件)。

7.10 停止程序：intr

Unix 提供了几种停止或者暂停程序的信号。这些信号是 **intr**、**quit**、**stop** 和 **susp**。我们将轮流讨论这些信号。一会儿您将看到，非常有趣的是，**stop** 信号并不是用来停止程序的信号。

在大多数系统上，**intr** 键是[^]C。但是在另一些系统上，使用的是<Delete>键。试一试，看看哪个键适合您的系统。

intr(interrupt, 中断)信号实际上有两个用途。首先，可以使用它停止一个僵死的程序。例如，假如您输入了一条命令，这条命令需要花费很长的时间才能结束，因此您决定不再等待而停止该命令。只需按下[^]C，这条远程命令将会终止，而您将返回到 shell 提示。

一些程序被编程为忽略 **intr** 信号。在这种情况下，程序总会提供一种明确定义的结束程序的方法(一些“quit”类型的命令)。通过忽略 **intr** 信号，程序可以防止您不小心按下[^]C 而导致程序故障。在这种情况下，我们称程序封闭(trap)了 **intr** 信号。

例如，假设您正在使用 **vi** 文本编辑器(第 22 章)编辑一个文件，如果您按下[^]C，会发生什么情况呢？**vi** 封闭了 **intr** 信号，因此它不会停止。为了停止该程序，您需要使用 **vi** 的退出命令。如果 **vi** 没有封闭 **intr** 信号，那么按下[^]C 将终止程序，还没有保存的数据将会丢失。

注意，有时候您可能看到将 **intr** 信号称为“break(中断程序运行)”键。如果您使用的是 PC 机，那么您可能知道 **^C** 充当 Microsoft Windows 下命令程序的中断程序运行键。可以看出，这一思想(还有许多其他思想)取自 Unix。

当您在 shell 提示处键入 Unix 命令时，**intr** 信号的第二个用途就出现了。如果键入了一条命令，而您改变了主意，可以按下 **^C** 取代 **<Return>**。按下 **^C** 将完全取消命令。

确定不要混淆了 **intr** 键(**^C**/**<Delete>**)和 **kill**(**^U**/**^X**)键。当键入一条命令时，**intr** 取消命令，而 **kill** 删除命令行上的所有字符。从效果上看，它们拥有相同的结果：正在键入的内容被删除，可以再输入一条新命令。

但是，只有 **intr** 停止程序。不管 **kill** 的名称如何，它不会真的杀死程序。

7.11 另一种停止程序的方法：quit

除了 **intr**(**^C**)之外，还有另外一种键盘信号 **quit**，可以用来停止程序。**quit** 键通常是 **^**<Ctrl-反斜线>**)。**

intr 和 **quit** 之间有什么区别呢？区别并不大。以前，**quit** 主要由需要终止测试程序的高级程序员使用。当按下 **^**<Ctrl-反斜线>**)时，它不仅停止程序，而且还会告诉 Unix 为此时内存中的内容制作一份副本。该信息存储在一个磁芯文件(core file)中，也就是一个命名为 **core**(计算机内存的老名称)的文件中。然后程序员可以使用特殊的工具分析磁芯文件，查找什么地方出了问题。**

现在，程序员拥有了更好的调试程序工具，因此，在大多数系统上，**quit** 信号不再生成磁芯文件，尽管一些编程环境仍在使用磁芯文件帮助调试。如果没有调试过程序，但是一个名为 **core** 的文件神秘地出现在您的一个目录中，那么这意味着您运行的程序出现严重的错误*而终止。除非真的需要这个文件，否则您可以删除它。实际上，您应该删除这个文件，因为 **core** 文件相当庞大，没有理由去浪费空间。

名称含义

磁芯文件

在计算机发展的初期，计算机内存由电子-机械式继电器构成，后来变成了真空管。1952 年，在 IBM 405 Alphabetical Accounting Machine 的实验版本中第一次应用了一个叫磁芯存储器(core memory)的新技术(405 是 IBM 公司的高端制表机，它是一个非常古老的设备，可以追溯到 1934 年)。

新类型的内存使用细小、圆形、中空的磁性设备(称为磁芯)，直径大约 0.25 英寸(6.4 毫米)(参见图 7-3)。大量的磁芯布置在用电线连接的网格上，电线连过每个磁芯。通过改变电线中的电流，可以修改单个磁芯的磁性，使之成为“off”或者“on”。这使二进制数据的存储和检索成为可能。

多年以来，磁芯存储器一直在不停地改进，到 20 世纪 60 年代，它成为 IBM 公司旗舰

* 最常见的原因是段故障(segmentation fault)。当程序试图访问不是为该程序分配的内存时就会发生这种故障，例如，不正确地使用指针(指针是指向其他东西的变量)。

产品 System/360 的主要部件。最终，随着技术的进步，磁芯存储器被半导体(晶体管)和集成电路取代，产生现在快速、高密度的存储芯片。但是，因为第一代现代计算机使用的是磁芯存储器，所以单词 core 成为“memory(存储器)”的同义词，成为一个延续至今的术语。

以前，程序调试是一个特别困难的过程，尤其是程序意外终止时。程序员使用的一种技术，就是使操作系统在程序终止时输出该程序使用的内存中的内容——这就是所谓的磁芯转储(core dump)，并且打印在纸上。磁芯转储需要使用许多张纸，而且需要很高的技能来解读。*

当 Unix 开发出来后，仍然使用这一技术。但是，Unix 将该数据保存在一个名称为 core 的特殊文件中，而不再是将磁芯转储打印在纸张上。如果您在测试程序，那么您可以通过按下 ^\ (quit 键) 强制程序终止，生成一个 core 文件。尽管以前的 Unix 程序员不得不学习如何解读磁芯转储文件，但是现在磁芯转储已经很少使用了，因为已经有更好的调试工具可用了。

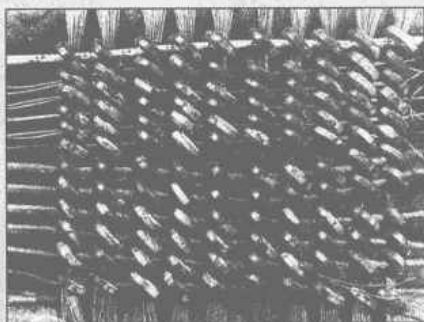


图 7-3 磁芯存储器

7.12 暂停显示: stop、start

当程序在屏幕的底部写一行输出时，其他行都要向上移动一个位置，该过程称之为向上滚动。如果程序生成输出太快，那么数据在阅读之前就滚动出屏幕了。

如果希望看一个这样的例子，可以使用下述两条命令中的一条。其中 **dmesg** 命令(第 6 章中已经遇到过)显示系统启动过程中的全部消息。另外一个命令是 **cat**，它显示 Termcap 文件的内容：

```
dmesg
cat /etc/termcap
```

cat 命令(将在第 16 章遇到)将数据连接起来，发送到默认输出位置上(或者“标准输出”)。在这个例子中，**cat** 将文件 **/etc/termcap** 中的数据复制到显示器上。但是，复制过程很快，大多数数据在阅读之前就滚动出屏幕了，这就是这个例子的目的。

在这种情况下，您有 3 种选择。首先，如果错过的数据不重要，那么您可以忽略这些数据。其次，可以重新启动生成数据的程序，并将该程序的输出发送到一个所谓的分页程

* 作为加拿大 Waterloo 大学的一名研究生，我是该大学计算中心的一名系统程序员，计算中心维护了两台 IBM 公司的大型机。有时候，我能看到一些非常有经验的系统程序员使用大量、多页面的磁芯转储来跟踪不易觉察的 bug。对我年青、单纯的眼睛来说，阅读磁芯转储是多么的神秘并令人畏惧！

序(例如 `less`, 参见第 21 章), 从而每次一屏地显示输出结果。在本章前面使用下述命令时我们就是这样做的:

```
less /etc/termcap
```

对于 `dmesg` 来说, 我们可以使用一个不同的命令来利用|(竖线)字符。这就是所谓的“管道符号”, 我们将在第 15 章中详细讨论它。管道的思想就是将 `dmesg` 的输出重新路由到 `less`。

```
dmesg | less
```

最后, 您可以按下 `^S` 键发送 `stop` 信号。这个信号告诉 Unix 临时停止屏幕显示。一旦显示过程暂停, 您可以通过按下 `^Q` 发送 `start` 信号重新启动屏幕显示。这两个信号容易记忆, 即“S”代表 Stop(停止), “Q”代表 Qontinue(也就是 `continue`, 即继续)。

`^S` 和 `^Q` 的使用相当便利。但是, 您应该理解 `^S` 只是告诉 Unix 停止输出的显示。它不会暂停正在执行的程序。程序继续执行, 不会停止生成输出。

Unix 将存储输出, 因此不会有输出丢失, 一旦您按下了 `^Q`, 剩下的全部输出将显示。如果在屏幕显示暂停时生成了大量的新数据行, 那么一旦按下了 `^Q`, 这些新数据行将飞速地冲过。

顺便说一下, 您可能会奇怪, 为什么选择 `^S` 和 `^Q` 来映射 `start` 和 `stop` 信号呢? 这看起来像一个古怪的选择。答案是, 在 Teletype ASR33 上, `<Ctrl-Q>` 发送 XON 码, 这个代码打开纸带阅读机; 而 `<Ctrl-S>` 发送 XOFF 码, 这个代码关闭纸带阅读机。

提示

如果您的终端曾经神秘地锁住, 那么可以试一试 `^Q`。您可能不小心地按下了 `^S`, 暂停了屏幕的显示。

当所有事情看上去都神秘停止时, 按下 `^Q` 不会造成任何伤害。

7.13 文件结束信号: eof

有时候, 您使用的程序期望您从键盘输入数据。当数据输入完, 没有数据再输入时, 可以通过按下 `^D` 发送 `eof`(end of file, 文件结束)信号指示这一点。

下面举一个例子: 在第 8 章中, 将讨论提供内置计算器服务的程序 `bc`。一旦启动了 `bc`, 就可以一个接一个地输入计算。在每个计算之后, `bc` 程序显示答案。当结束计算时, 按下 `^D` 告诉 `bc` 程序没有数据了。在接收到 `eof` 信号之后, 程序就会终止。

7.14 shell 和 eof 信号

在第 2 章中, 解释过 shell 是读取 Unix 命令并解释命令的程序。当 shell 准备好读取命令时, 它显示一个提示。在这个提示下, 可以键入命令并按 `<Return>` 键。在按下 `<Return>`

键之后, shell 处理命令, 然后显示一个新的提示。在一些情况下, 命令可能启动一个程序, 例如文本编辑器, 您可能会使用这个程序一段时间。当结束程序时, 您将返回到 shell 提示。

因此, 一般来说, 提供 CLI(命令行界面)的 Unix 会话由一条又一条命令的输入构成。

尽管 shell 看上去似乎神秘, 但是它实际上只是一个程序。而且从 shell 的角度来看, 键入的命令只是需要它处理的数据。因此, 可以通过指示已经没有数据需要处理而停止 shell。换句话说, 就是可以通过按下 **^D**(eof 键)停止 shell。

但是停止 shell 真正意味着什么呢? 它意味着您已经结束了工作, 而当 shell 停止时, Unix 会自动将您注销。这就是为什么按下 **^D** 可以注销系统的原因。您实际上是在告诉 shell(和 Unix)已经没有工作需要做了。

当然, 这里有一个潜在的问题。如果您不小心按下了 **^D** 会发生什么情况呢? 您将被立即注销。解决办法就是告诉 shell 封闭 eof 信号。这一点如何做取决于您正在使用什么 shell。下面轮流说明各个 shell 的操作, 具体包括 Bash、C-Shell 和 Korn Shell, 您可以使用自己的特定 shell 进行实验。

7.15 Bash: 封闭 eof 信号

Bash 是 Linux 的默认 shell。为了告诉 Bash 忽略 eof 信号, 需要使用一个叫 **IGNOREEOF** 的环境变量(注意在这个单词中有两个连续的 **E**, 因此在拼写时一定要仔细)。下面解释它的工作方式。

IGNOREEOF 被设置成一个特定的数字, 用来指定在注销之前 Bash 会忽略特定行开头的 **^D** 多少次。在设置 **IGNOREEOF** 时, 需要使用一个类似于下面的命令(您可以使用任意希望的数字来取代 5)。

```
IGNOREEOF=5
```

为了测试该命令, 可以重复按 **^D**, 统计在注销之前要按多少次 **^D**。

当设置了 **IGNOREEOF** 之后, 如果按下 **^D**, 那么将看到一个告诉您不能通过按 **^D** 进行注销的消息。如果使用的是登录 shell(也就是在登录时自动启动的 shell), 将会看到如下消息:

```
Use "logout" to leave the shell.
```

如果使用的是一个子 shell(也就是说一个在登录后启动的 shell), 将会看到如下消息:

```
Use "exit" to leave the shell.
```

如果基于某些原因, 希望关闭 **IGNOREEOF** 特性, 那么只需将它设置为 0 即可:

```
IGNOREEOF=0
```

显示 **IGNOREEOF** 的当前值, 使用下述命令:

```
echo $IGNOREEOF
```

为了在每次登录时自动设置 **IGNOREEOF**，需要在 **.profile** 文件中放置一条合适的命令(参见第 14 章)。

7.16 Korn Shell: 封闭 eof 信号

Korn Shell 是多种商业 Unix 系统的默认 shell。另外，FreeBSD 的默认 shell 几乎完全与 Korn Shell 相同。

为了告诉 Korn Shell 忽略 **^D**，需要设置一个叫 **ignoreeof** 的 shell 选项(注意在这个单词中有两个连续的 **e**，因此在拼写时一定要仔细)。设置该选项的命令如下：

```
set -o ignoreeof
```

一旦设置了 **ignoreeof** 选项，如果按下 **^D**，将看到一个告诉您不能通过按下 **^D** 进行注销的消息：

```
Use "exit" to leave shell.
```

如果基于某些原因，希望关闭 **ignoreeof** 选项，则可以使用下述命令：

```
set +o ignoreeof
```

显示 **ignoreeof** 的当前值，使用下述命令：

```
set -o
```

这将显示所有的 shell 选项，并说明它们是打开的还是关闭的。

为了在每次登录时自动设置 **ignoreeof** 选项，需要在 **.profile** 文件中放置一条合适的 **set** 命令(参见第 14 章)。

7.17 C-Shell: 封闭 eof 信号

为了告诉 C-Shell 忽略 **^D**，需要设置一个叫 **ignoreeof** 的 shell 变量(注意在这个单词中有两个连续的 **e**，因此在拼写时一定要仔细)。设置该变量的命令如下：

```
set ignoreeof
```

一旦设置了 **ignoreeof** 变量，如果按下 **^D**，将会看到一个告诉您不能通过按下 **^D** 进行注销的消息。如果使用的是登录 shell(也就是在登录时自动启动的 shell)，将会看到如下消息：

```
Use "logout" to logout.
```

如果使用的是一个子 shell(也就是在登录后启动的 shell)，将会看到如下消息：

```
Use "exit" to leave csh.
```

(**cs**h 是 C-Shell 程序的名称。)

如果基于某些原因, 希望关闭 **ignoreeof** 特征, 则可以使用下述命令:

```
unset ignoreeof
```

显示 **ignoreeof** 的当前值, 使用下述命令:

```
echo $ignoreeof
```

如果设置了 **ignoreeof**, 则看不到什么内容。如果没有设置 **ignoreeof**, 将看到:

```
ignoreeof: Undefined variable.
```

为了在每次登录时自动设置 **ignoreeof** 变量, 需要在 **cs**hrc 文件中放置一条合适的 **set** 命令(参见第 14 章)。

7.18 显示键映射: stty -a

到目前为止, 已经提到许多键盘信号, 每个键盘信号都对应于键盘上的一些键。这些键盘信号如图 7-4 所示。本书所示范的这些键映射都是最常见的键映射, 但是它们是不固定的。

信号	键	作用
erase	<Backspace>/<Delete>	删除键入的最后一个字符
werase	^W	删除键入的最后一个单词
kill	^X/^U	删除整行
intr	^C	停止正在运行的程序
quit	^\	停止程序并保存 core 文件
stop	^S	暂停屏幕显示
start	^Q	重新启动屏幕显示
eof	^D	指示已经没有数据

图 7-4 重要键盘信号一览表

显示系统的键盘映射时, 使用下述命令:

```
stty -a
```

stty 是 “set terminal(设置终端)” 命令, **-a** 意味着 “显示所有的设置”。

stty 命令显示了若干行关于终端的信息。我们感兴趣的是那些显示键盘信号及它们所映射的键的信息行。下面是一个 Linux 系统的例子:

```
intr = ^C; quit = ^\; erase = ^?; kill = ^U;
eof = ^D; eol = <undef>; eol2 = <undef>;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
```



```
werase = ^W; lnext = ^V; flush = ^O;
```

下面是一个 FreeBSD 系统的例子：

```
discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
eol2 = <undef>; erase = ^?; erase2 = ^H; intr = ^C;
kill = ^U; lnext = ^V; quit = ^\; reprint = ^R;
start = ^Q; status = ^T; stop = ^S; susp = ^Z; werase = ^W;
```

注意在 FreeBSD 系统示例中有一个 **erase2** 信号。

可以看出，这里有几个信号本书还没有提到过。这些信号中，大多数对于日常的工作并不重要，因此可以忽略它们。

提示

在第 26 章中，将讨论如何暂停与重新启动正在运行的程序。那时，您将看到通过按 **^Z** 键可以暂停程序，**^Z** 映射到 **susp**(suspend, 挂起)信号上。一旦使用 **^Z** 暂停了一个程序，程序就会停止运行，直至通过输入 **fg**(foreground, 前台)命令重新启动程序。

因此，如果您正在工作，突然之间，程序停止了，而且您看到一个类似于 **Suspended** 或者 **Stopped** 的消息，那么这意味着您不小心按下了 **^Z** 键。

当这种情况发生时，您所需做的事情就是输入 **fg** 命令，这样程序就会恢复运行。

7.19 修改键映射：stty

如果希望修改键映射，可以使用 **stty** 命令。只需键入 **stty**，后面跟着信号的名称，然后是新的键赋值即可。例如，将 **kill** 键修改为 **^U** 的命令如下：

```
stty kill ^U
```

重点在于：一定要确保以两个单独的字符的形式键入 **<Ctrl>** 键组合，而不是一个真正的 **<Ctrl>** 组合键。**stty** 将会指出其中一个字符代表 **<Ctrl>** 键。例如，在这个例子中，应该键入 **^**(插入记号)字符，后面跟一个 **U** 字符，而不应该键入 **<Ctrl-U>**。

当在 **stty** 命令中使用带 **<Ctrl>** 的字符名时，不必键入大写字母。例如，下述两条命令都能正常工作：

```
stty kill ^u
stty kill ^U
```

只需记住，要键入两个单独的字符。

严格地讲，可以将任意键映射到一个信号上。例如，可以将字母 **K** 映射到 **kill** 信号上。下面两条命令都可以完成上述工作：

```
stty kill k
stty kill K
```

当然，这样的映射只会导致问题。每次按下 **<K>** 键时，Unix 都将删除刚键入的那一行

字符！对朋友玩这样一个把戏肯定十分有趣！*

通常，在映射中只使用<Ctrl>组合键。实际上，在几乎全部情况中，最好是保持事情原来的方式，坚持标准的键赋值。

但是，还有一种情况，在这种情况下，您可能希望进行修改。假设您经常通过网络连接一台远程主机，在这台主机上，**erase** 键是`^?`。但是，您的 **Backspace** 键发送的是`^H`。为了使操作更加方便，需要将`^H`映射到 **erase**。这样就允许您在按下<Backspace>键时删除一个字符。

```
stty erase ^H
```

下面举一个相反的例子。您连接到一台主机，这台主机将`^H`映射到 **erase**。但是，您的<Backspace>(或者<Delete>)键发送`^?`信号。这时，使用 **stty** 修改映射的命令如下所示：

```
stty erase ^?
```

记住，表示法`^?`并不是指一个实际的<Ctrl>键组合。`^?`只是一个两字符的缩写，代表“whichever key on your keyboard sends the DEL code(键盘上用来发送 DEL 代码的任意键)”。

如果决定查看键盘的映射情况，可以使用前面描述的命令进行查看：

```
stty -a
```

另外，还可以只输入 **stty** 命令本身：

```
stty
```

这将显示一个简短报告，仅显示那些默认值已经改变的映射。

7.20 命令行编辑

当在命令行上键入时，光标指向下一个可用位置。每键入一个字符，光标就向右移动一个位置。

在键入过程中，出错了该怎么办呢？正如本章前面讨论的，您可以按<Backspace>键，删除一个或多个字符，然后再键入新字符。

但是，如果希望修复命令行开头的错误，而且您已经键入了 20 个字符，该怎么办呢？当然您可以按<Backspace>键 21 次，修复错误，然后重新键入 20 个字符。但是，还有一种更简单的方法。

对于大多数 shell(并不是全部)来说，可以简单地使用左箭头键(以后称之为<Left>)。每按一次这个键，光标就会在不删除任何内容的情况下向左移动一个位置。然后就可以进行希望的修改，并按<Return>键运行命令。

试一试下面的例子，这个例子使用的命令是 **echo**(**echo** 命令简单地显示赋予它的内容)。

* 希望您没有看到这句话！

键入下述命令：

```
echo "This is a test!"
```

现在按<Return>键。shell 将显示 “**This is a test!**”。现在键入下述内容，但是不按<Return>键：

```
echo "Thus is a test!"
```

在按<Return>键之前，需要将 **Thus** 修改为 **This**。此时，光标应该位于这一行的末尾，因此重复按<Left>键，直至光标正好位于 **Thus** 中 **u** 的右边。按<Backspace>键一次，删除 **u**，然后键入 **i**。现在可以按<Return>键了，您将会看到正确的输出。

这就是**命令行编辑**(command line editing)的一个例子，也就是说，在将命令行上的内容发送给 shell 之前，修改命令行中的内容。注意在按<Return>键之前，不必再将光标移至命令行的末尾。

提示

当按下<Return>键时，命令行上的字符被发送给 shell 进行解释。因为光标不生成字符，所以 shell 不关心在向它发送命令时光标所在的位置。

这意味着，当进行命令行编辑时，可以从这一行的任何位置上按<Return>键。光标不必位于命令行的末尾。

所有现代的 shell 都支持一些类型的命令行编辑，但是各个 shell 之间命令行编辑的细节各不相同。基于这一原因，我们将在本书后面讨论每种 shell 时，再对命令行编辑做进一步讨论。现在，将讲授 3 个最重要的技术。您可以试一试，看看这 3 个技术是否适合您的特定 shell。

首先，在键入时，可以使用<Left>和<Right>箭头键在命令行中移动光标。上述例子就使用了这种方式。

其次，在任何时间，都可以按<Backspace>键删除前一个字符。对于一些 shell 来说，还可以使用<Delete>键删除当前字符(这里所讲的<Delete>键指 PC 机键盘上的<Delete>键，它紧挨着<Insert>键)。

第三，在输入命令时，shell 将把命令保存在一个不可见的“历史列表”中。可以使用<Up>和<Down>箭头键在历史列表中向后或者向前移动查看命令。当按<Up>键时，当前命令消失，被前一条命令取代。如果再次按<Up>键，就会得到后者的前一条命令。

因此，可以按<Up>键多次，恢复前面的命令。如果发现返回得太多，则可以通过按<Down>键在列表中向下移动。在选择了命令之后，还可以编辑命令行，并且通过按下<Return>键重新向 shell 提交命令。

名称含义

破坏性退格、非破坏性退格

当按下<Backspace>键时，它使光标向左移动一个位置，并且删除一个字符。当按下<Left>箭头键时，它使光标向左移动一个位置，但是不删除字符。

在某种意义上，这两种动作相似，因为它们都可以使光标向后移动。它们之间的唯一区别就是字符是否被删除。为了强调这一思想，我们使用了两个术语：“破坏性退格”和“非破坏性退格”。

破坏性退格(destructive backspace)指的是当光标向后移动时，光标经过的字符被删除。当按<Backspace>键时就发生这种情况。

非破坏性退格(non-destructive backspace)指的是当光标向后移动时，没有内容被修改。当按<Left>键时就发生这种情况。

7.21 返回和换行

本章前面，我们讨论了 Unix 处理<Backspace>键的方式可以追溯到最初的 Unix 终端 Teletype ASR33。具体而言，Teletype ASR33 在删除纸带上的字符时包含有两种 Teletype 码 (BS 和 DEL)。Unix 开发人员选择其中的一个代码用于 **erase** 信号。

更有趣的是，当轮到决定<Return>键的操作内容时，他们都选择了相同类型的选项。此外，事实证明他们有关<Return>键的决定要比对<Backspace>键的决定更加重要。这是因为他们选择的代码，不仅用于<Return>键，而且还是文本文件每行结束的一个特殊标记。为了开始我们的讨论，需要再次返回到 Teletype ASR33 的那个时代。

Teletype ASR33 有一个打印头，使用色带在纸上打印字符。在字符打印时，打印头从左向右移动。当打印头移动到一行的末尾时，必须进行两件事情。首先，打印纸需要向上移动一行；其次，打印头(依附在“托架”上)必须返回到最左边。

为了使 Teletype 执行这些操作，需要在打印的数据中植入代码。而数据可能来自于键盘，也可能来自于通信线路，或者来自于纸带阅读机。

第一个代码是 CR(carriage return, 托架返回)将托架返回到最左边的位置上。第二个代码是 LF(linefeed, 换行)，使打印纸向上移动一行。因此，序列 CR-LF 执行打印一个新行所需的准备动作。

在键盘上，按下<Return>键或者[^]M(它们等价)可以发送一个 CR 码。按下<Linefeed>或者[^]J 键可以发送 LF 码(如果查看本章前面的图 7-1，那么就会发现<Return>键位于从顶部起第二排的最右边。<Linefeed>键位于<Return>键左边一个位置处)。

当 Unix 开发人员开始使用 Teletype 作为终端时，他们基于 CR 和 LF 码创建了两个信号。CR 码变成了返回信号。LF 码变成了换行信号。

现在，我们先问一个问题：当在 Unix 终端上键入时，按下<Return>键会发生什么事情呢？在回答这个问题之前，需要先解释一下 Unix 如何将纯文本组织成文件。

7.22 新行字符的重要性

正如前面讨论的，Unix 使用两个基于老式的 Teletype 的信号：返回和换行。在键盘上，

按下[^]M 将发送返回信号, 按下[^]J 将发送换行信号。

既然返回和换行其实与[^]M 和[^]J 相同, 所以我们通常称它们为字符, 而不是信号。下面是使用它们的 3 种不同场景。

第一: 当文件中包含有文本数据时, 通常将数据分成行。在 Unix 中, 使用[^]J 字符标记每行的结束。当以这种方式使用[^]J 时, 我们称它是**新行字符**, 而不是换行字符。因此, 在程序从文件中读取数据时, 当程序遇到一个新行字符(也就是一个[^]J 字符)时, 程序就知道它已经到达了一行的末尾。

第二: 当在终端上键入字符时, 在行的末尾按下<Return>键。这样做可以发送返回字符, 也就是[^]M。

第三: 当数据显示时, 每次向终端发送一行字符。在每行的末尾, 光标必须移动到下一行的开头。和 Teletype 一样, 这包括两个单独的动作: 一个“托架返回”动作将光标移动到行的开头, 后面跟一个“换行”动作将光标移动到下一行。对于“托架返回”动作, Unix 发送一个返回字符(也就是[^]M)。对于“换行”动作, Unix 发送一个换行字符(也就是[^]J)。因此, 当数据显示时, 每行必须以[^]M[^]J 结束。

Unix 最一流的特性之一就是 will 将键盘键入的数据视为与从文件中读取的数据相同。例如, 假如您有一个程序需要读取一系列的名称, 每行一个。这样的程序既可以从磁盘上的文件中读取名称, 也可以从键盘输入读取名称。该程序不需要以特殊的方式编写就可以拥有这样的灵活性。这一特性称为“标准输入”, 已经构建在 Unix 中。标准输入允许所有的 Unix 程序以相同的方式读取数据, 而不必考虑数据源(我们将在第 15 章中详细讨论这一思想)。

为了使标准输入正常工作, 每一行数据都必须以一个新行字符结束。但是, 当使用键盘键入字符时, 字符以一个返回字符结束每行, 而不是新行字符。这就产生了一个问题。

同理, 当 Unix 程序输出数据时, 它们可以利用“标准输出”。这允许所有的程序以相同的方式写数据, 而不必担心数据写到哪里去。

当数据写到文件中时, 每行必须以一个新行字符(也就是[^]J)结束。但是, 当数据写到终端上时, 每行必须以返回+新行([^]M[^]J)结束。这就产生了第二个问题。

这些问题可以用两种方式解决。第一种, 在键入过程中, 无论何时, 当按下<Return>键时, Unix 都将返回字符改变为新行字符。也就是说, Unix 将[^]M 改变为[^]J。

第二种, 当数据要写到终端上时, Unix 将每个新行字符改变为返回字符+新行字符。也就是说, Unix 将[^]J 改变为[^]M[^]J。

为了便于领会, 下面给出一个简短的小结:

(1) 返回字符=[^]M。

(2) 换行字符=新行字符=[^]J。

(3) 一般而言, 每行文本必须以一个新行字符结束。

(4) 当按下<Return>键时, 将发送一个返回字符, Unix 自动地将返回字符改变为新行字符。

(5) 在终端上显示数据时, 每行必须以字符序列“返回+新行”结束。因此, 当数据从文件发送到终端显示时, Unix 自动地将每行末尾的新行字符改变为返回字符+换行字符。

刚开始时, 这看上去可能有点使人迷惑。最终, 您将会明白所有这些都非常合理, 这时候, 您就会知道自己最终开始思考 Unix 了。

提示

在文本文件中，Unix 使用 `^J`(newline) 字符标记每行结束。但是，Microsoft Windows 的做法不同。Windows 使用 `^M^J` 标记每行的结束(在 Unix 术语中，这将是返回字符+换行字符)。

因此，当从 Unix 中向 Windows 中复制文本文件时，每个 `^J` 必须改变为 `^M^J`。相反，当从 Windows 中向 Unix 中复制文本文件时，每个 `^M^J` 必须改变为 `^J`。

当使用程序在两台这样的计算机之间复制文件时，程序应该知道如何自动地进行这种改变。如果程序不能自动进行改变，则可以使用实用工具程序完成该工作。

7.23 `^J` 的一个重要用途：stty sane、reset

除非您是一名程序员，否则您并不真的需要掌握与返回字符和新行字符相关的全部技术细节。您只需记住在每行的末尾按下 `<Return>` 即可，其他的让 Unix 去做。

但是，在某些情形中理解这些思想非常有帮助。在一些罕见情况中，终端的设置可能将终端搞乱，从而不能正常地工作。在这种情况下，有两条命令可以用来将终端设置恢复为合理值，这两条命令是 `stty sane` 和 `reset`。

在一些罕见的情况中，您可能发现当您试图按下 `<Return>` 键输入这些命令中的一个时，返回字符不能向新行字符正常转换，Unix 不接收命令。如果发生了这种情况，只需简单地按下 `^M` 取代 `<Return>` 即可。这样就会有效，因为这两个键实质上相同。

解决办法就是按下 `^J`(同新行字符)，而 `^J` 是所有 Unix 都接受的。因此，当其他所有方法都失效时，键入下述命令之一可以复原终端。一定要确保在命令前和命令后键入 `^J`。如果您希望，现在就可以试一试，这两条命令不会破坏任何东西。

```
<Ctrl-J>stty sane<Ctrl-J>
<Ctrl-J>reset<Ctrl-J>
```

您可能会问，如果是这种情况，那么可以在任何时间按下 `^J` 来取代 `<Return>` 输入一条命令吗？当然可以——您也可以试一试。

为了证明这些命令是多么有用，下面讲述一个真实的故事。

我有一个朋友叫 Susan，有一天她在帮助别人安装 Linux。他们使用了一个程序，这个程序允许选择在内核中包含什么选项。该程序需要一个特定的目录，但是这个目录还不存在，因此 Susan 按下 `^Z` 暂停程序。她现在有机会创建该目录了。

但是，碰巧的是该程序为了防止改变显示禁用了返回字符的作用。这意味着，无论 Susan 何时输入命令，输出都不正常显示。

(记住，当 Unix 将数据写到终端时，它将返回字符+换行字符放在每行的末尾。您可以想一想，当只有换行字符起作用时，会发生什么情况呢？)

但是，Susan 也不简单。她输入了 `reset` 命令，很快，终端工作正常了。然后她创建了需要的目录，重新启动程序的安装过程，从而顺利地安装了程序。

7.24 程序员和公主的神话

很久很久以前，有一位年青、英俊、迷人的程序员(您知道这是一个神话)，他赢得了一位漂亮公主的爱情。但是，在他们婚礼的前夜，公主被绑架了。

幸运的是，公主镇定自若地用她项链的珍珠留下了痕迹。程序员追随着痕迹来到了一个遥远的角落——法律所不能及的硅谷(Silicon Valley)，在这里，他发现了他的爱人被市场部邪恶的副经理(Vice President, VP)绑架在一个废弃的技术支持中心内。

快速地思考之后，程序员拿出一块强大的磁铁，进入这个建筑物内。他发现了公主，并破门进入房间。在这个房间中，市场部的 VP 站在那里，正在沾沾自喜地注视着害怕的公主。

“立即放开那个女孩，”程序员咆哮地说，“否则我将使用这块磁铁搞乱你所有的磁盘。”

VP 按下一个秘密的按钮，一眨眼的工夫，4 个更加丑陋笨重的副经理进入房间。

“从另一方面来说，”程序员说，“或许我们可以做个交易。”

“你有什么主意？”VP 说。

“你可以给我设置任何 Unix 任务。”程序员回答到，“如果我能做出来，公主和我获得自由。如果我失败了，那么我将离开，永远不回来，而且公主就是你的了。”

“没有问题。”VP 说，他的眼睛像板油环绕的两个谄媚的红色牛轧糖一样闪光，“坐在这台终端前面。你的任务有两部分。首先，使用一条单独的命令显示当前的时间和日期。”

“小孩子的游戏，”程序员说，他键入了 `date` 并按下了 `<Return>` 键。

“现在，”VP 说，“再做这个问题。”但是，当程序员再次键入 `date` 时，VP 说道：“但是这次不能使用 `<Return>` 键和 `^M`。”

“你输入，你这个愚蠢的小丑！”程序员叫喊到，随之他按下了 `^J`。公主夺回来了，然后他将公主带到正在等待他们的 Ferrari(一种汽车，译者注)，从此他们过着自由的生活。

7.25 练习

1. 复习题

1. 为什么 Unix 习惯于使用缩写“`tty`”指代终端呢？
2. 为什么 Unix 习惯使用单词“`print`”指代在显示器上显示数据呢？
3. 术语“`deprecated`(不推荐)”意味着什么呢？
4. Unix 如何知道您正在使用哪个终端？
5. 按哪个键删除刚刚键入的最后一个字符？删除最后一个单词呢？删除整行呢？

2. 应用题

1. 默认情况下, **erase** 键就是<Backspace>键(或者是<Delete>键, 在 Macintosh 机上)。通常这个键映射到[^]H, 或者[^]? (不经常)。使用 **stty** 命令将 **erase** 键改变为大写字母 “X”。一旦这样做了之后, 按下 “X” 就可以删除刚键入的最后一个字符。测试一下, 当按下一个小写的 “x” 时会发生什么情况? 为什么呢? 现在使用 **stty** 将 **erase** 键修改回<Backspace>(或者<Delete>)键。测试一下, 确保工作正常。

3. 思考题

1. 一种注销系统的方式就是在 shell 提示下按[^]D(发送 eof 信号)。因为可能会不小心地按下[^]D, 所以您可以告诉 shell 忽略 eof 信号。为什么这不是默认情况呢? 这可以说明使用 Unix 的人具有什么样的特征呢?

2. 在第 1 章中, 提到过 Unix 的第一版由 Ken Thompson 开发, 其目的是他可以运行一个叫 Space Travel 的游戏。在本章中, 我解释到第一个使用 Termcap(终端信息数据库)和 curses(终端管理器界面)的程序是一个基于文本的幻想游戏 Rogue, 它由 Michael Toy 和 Glenn Wichman 开发。创建一个新的操作系统以及试验一组全新的界面都是相当耗时、困难重重的任务。激发 Thompson 以及后面的 Toy 和 Wichman 从事这样有挑战性的工作的原因看上去是如此的微不足道, 您是怎么想的? 如果您管理着一组程序员, 那么您认为什么样的激励会得到他们的积极响应呢(除了金钱)?



能够立即使用的程序

当在 shell 提示处输入命令时，实际上是告诉 shell 根据这个名称运行程序。例如，当输入 **date** 命令时，就是请求 shell 运行 **date** 程序*。

Unix 差不多有数千个不同的程序，这意味着可以输入数千条不同的命令。其中许多程序要求理解一些理论。例如，在使用文件系统命令之前，需要学习一些文件系统的知识。还有的程序非常复杂，需要大量的时间才能掌握。例如，Unix 的两个主要文本编辑器(**vi** 和 **Emacs**)就是这种情况。这些程序非常有用，但是也相当复杂。实际上，在本书中，有一整章的内容是专门介绍 **vi** 的。

但是，还有一些程序不需要特殊的知识，而且使用起来也不是特别困难。这些程序是能够立即使用的程序，在本章中，将介绍一些这样的程序。

在众多的 Unix 程序中，我挑选了一些自己觉得特别有用、有趣或者搞笑的程序。我有两个目的。首先，我希望您知道这些程序，因为它们确实都非常有用、有趣或者搞笑。其次，本书前面已经介绍了不少的通用理论，我希望给您一个熟悉使用 Unix CLI(命令行界面)的机会。传统上讲，人们学习 Unix 的方式之一就是寓学于乐。下面让您和我继续保持这一传统。

8.1 在系统中查找程序：which、type、whence

正如第 2 章中所讨论的，Unix 并不是指一个具体的操作系统。它是一族操作系统的总称：Linux、FreeBSD、Solaris、AIX 等。此外，即便 Linux 本身也不是指一个系统。名称“Linux”指的是任何使用 Linux 内核的 Unix 系统，世界上差不多有数百种不同的 Linux 发行版。

尽管各种不同的 Unix 拥有许多共同之处，但是它们并没有提供完全相同的程序集。例如，伯克利 Unix(FreeBSD、NetBSD 和 OpenBSD)中存在的一些程序，在 Linux 系统中就不存在。此外，即使比较两台运行相同版本 Unix 的计算机，您也会发现不同的程序。这

* 严格地讲，事情并非完全如此。一些命令是“嵌入”到 shell 中的，这意味着它们实际上并不是单独的程序。

但此时，单独程序和内置程序之间的区别并不重要。

是因为，在 Unix 安装过程中，有许多关于安装哪些程序的选项，并不是每个人都会选择相同的选项。

基本的 Unix 就是基本的 Unix，因此大多数系统都安装了全部重要的程序。然而，不管您使用的是何种类型的 Unix，本章(或者本书其他地方)中的程序也有可能没有安装在您的系统上。

基于这一原因，现在我希望首先考虑两个重要的问题：您如何知道一个特定的程序是否在您的系统上存在？如果系统上没有这个程序，那么您应该怎么做？

查看是否可以访问一个程序的最简单的方法就是在 shell 提示处键入该程序的名称(记住，shell 的任务就是解释命令)。如果系统上有这个程序，则将会发生一些事情。如果系统上没有这个程序，那么 shell 将会说明它找不到这个程序。不要担心这种方式的试验结果。如果输入了一条不存在的命令，那么它不会产生任何问题——除了产生一个错误消息。

查看某个程序是否可用的一种更精确的方法就是使用 **which** 命令。**which** 的目的就是让 shell 回答下述问题：如果我准备输入一条具体的命令，那么将会运行哪个程序？如果这个问题有答案，那么这个程序就安装在您的系统上，也就是说可以使用这个命令。如果这个问题没有答案，那么这个命令在您的系统上不可用。

在使用 **which** 时，只需在键入的 **which** 命令后跟一个或者多个程序的名称即可，例如：

```
which date
which date less vi emacs
```

下面是第一条命令的输出：

```
/bin/date
```

在这个例子中，**which** 命令告诉您，如果输入了 **date** 命令，那么 shell 将运行存储在 **/bin/date** 文件中的程序。如果还不理解 Unix 文件系统(我们将在第 23 章中讨论)，那么这个名称可能对您来说还没有意义。但是不用担心，现在重要的事情是 **which** 已经找到了一个可以运行的程序，这就告诉您，在您的系统上 **date** 是一条有效的命令。

如果询问 **which** 一个在系统上不存在的程序会发生什么事情呢？例如：

```
which harley
```

这条命令的结果有两种可能性，这取决于 **which** 的版本。第一种是，不发生任何事情。也就是说，没有任何输出。这意味着 **which** 找不到指定的程序。这是许多 Unix 程序的特征：如果没有什么可说的，那么它们就什么也不说(但愿更多人拥有这样的理念)。

第二种可能性是您将看到一个错误消息。下面是一个典型的错误消息：

```
/usr/bin/which: no harley in (/usr/local/bin:/usr/bin:
/bin:/usr/X11R6/bin:/home/harley/bin)
```

which 告诉您，它在您的搜索路径中找不到一个名为 **harley** 的程序。当讨论各种 shell 时，我们将讨论搜索路径。现在重要的是要意识到，因为 **which** 找不到一个名为 **harley** 的程序，所以不能在您的系统上使用 **harley** 命令。

那么如果您希望试着运行本章中的一个程序，而该程序看上去似乎不在您的系统上，

该怎么办呢？十分明显，第一件事情就是检查拼写。聪明人有时候也会犯拼写错误，您和我也不例外*。

如果您已经正确拼写了命令，但它还是找不到，那么您有几种选择。首先，您可以忘掉它。本章有许多程序可供尝试，其中大多数程序在您的系统上可用。

其次，如果能访问其他 Unix 系统，那么您可以在这些 Unix 系统上尝试一下，看看它们是否拥有该命令。

第三，如果使用的是共享系统，那么您可以请求系统管理员，看看他们是否愿意在系统上安装该程序。系统管理员是非常忙的人，但是，对一个有知识的人来说，安装一个新程序并不会花太长的时间，所以如果您礼貌殷勤，应该会得到想要的帮助。

最后，如果使用自己的 Unix 计算机，那么您可以自己安装程序。程序的安装细节对于不同的 Unix 系统各不相同，而且也超出了本章的讨论范围。或许您可以找一个身边的高手，让他帮您安装程序。

如果您使用的 shell 是 Bash(参见第 12 章)，那么 **which** 命令还有一个备用命令 **type**，例如：

```
type date
```

如果使用的是 Korn shell(参见第 13 章)，那么您可以使用 **whence** 命令：

```
whence date
```

type 和 **whence** 命令有时候要比 **which** 命令显示更多的细节信息。这在特定的环境下很有用。但是，在实际应用上，还是 **which** 命令的应用最广泛。

在结束本节之前，我先询问一个问题，如果对 **which** 命令自身使用 **which** 命令，您认为会发生什么事情呢？换句话说，就是 **which which** 命令将得到什么结果呢？当您有时间时，可以试一试：

```
which which
```

如果使用的 shell 是 Bash，还可以试一试下述命令：

```
type which
which type
type type
```

8.2 如何停止程序

大多数 Unix 命令执行一项任务，然后自动停止。例如，在下一节中，我们将讨论 **date** 命令。该命令所做的工作就是显示时间和日期。然后它停止运行，您又返回到 shell 提示。

有一些命令比较复杂。当启动它们时，它们就将您置于一个环境中，在这个环境中，

* 实际上，鉴于您和我比大多数人都聪明，所以我们更有可能犯较多的拼写错误，因为我们的思想转得比较快。

可以通过一条接一条地输入命令与程序进行交互。当结束该程序的工作时，需要输入一个特殊的命令来退出程序，此时程序停止，您又返回到 shell 提示。

这一命令类型的一个例子是 **bc**，**bc** 是本章后面即将讨论的一个计算器程序。当使用 **bc** 时，需要先启动它，然后就可以一次输入一个计算命令。只要愿意，可以运行该程序任意长的时间。当结束计算时，必须告诉 **bc** 您准备退出。为此，要输入 **quit** 命令。

对于这样的程序，一般总是有一个特殊的命令来退出程序。该命令通常是 **q** 或者 **quit**。为了确定退出命令的名称，您可以阅读 Unix 的联机手册(将在第 9 章讨论)中有关该程序的文档。

除了使用特殊的退出命令之外，还有一种简单停止程序的方式，即告诉程序已经没有数据需要处理了。这种方法是通过按 **^D**(**<Ctrl-D>**)，即 **eof** 键(参见第 7 章)实现的。这通常是停止程序的最简单方法。例如，当使用 **bc** 程序时，只需输入 **^D** 就可以使程序停止。

如果其他所有方法都失败了，那么您还可以按 **intr** 键停止程序，**intr** 键可能是 **^C** 或者 **<Delete>**(参见第 7 章)。

提示

大多数程序拥有自己的可交互环境，提供了一些内置帮助。为了访问该帮助，可以试一试键入 **help** 或者 **h** 或者 **?(问号)**。

8.3 显示时间和日期：date

date 命令是所有 Unix 命令中最重要的命令之一。简单地输入：

```
date
```

Unix 将显示当前的时间和日期。下面是一个样本输出，注意 Unix 使用的是 24 小时制时钟。

```
Sun Dec 21 10:45:54 PST 2008
```

如果您生活在一个使用夏令制时间的地方，那么 Unix 知道如何在春天向前以及秋天向后调整合适的时间。这里示范的例子显示的是太平洋标准时间(Pacific Standard Time)。在夏天的时候，它将是太平洋夏令制时间(Pacific Daylight Time)。

注意，**date** 同时显示时间和日期，因此当希望知道时间时，这就是要使用的命令。Unix 系统中还有一个 **time** 命令，但是这个命令并不显示时间。它测量程序的运行时间。

从本质上讲，Unix 并没有运行在本地时间上。所有的 Unix 系统都使用协调世界时(Coordinated Universal Time, UTC)，它是格林威治标准时间(Greenwich Mean Time, GMT)的现代名称。Unix 在需要时默默地在 UTC 和本地时区之间进行转换。本地时区的细节信息在安装 Unix 时指定。

有时候，查看 UTC 时间比较便利。为了显示 UTC 时间，只需使用：

```
date -u
```

您将看到一个类似于下述形式的时间和日期：

```
Sun Dec 21 18:45:54 UTC 2008
```

顺便说一下，该时间是上一个例子的 UTC 等效时间。

有关时间的更多信息，请参见附录 H，在附录 H 中，将解释时区、24 小时制和 UTC，还将示范如何从一个时区转换到另一个时区，并使用几个简单的例子进行描述。

8.4 显示日历：cal

Unix 的一个出色之处就是它不是由一个委员会设计的。当 Unix 程序员决定需要新工具时，他就可以自己编写程序，并将程序添加到系统中。这种情况的一个最好例子就是 **cal** 命令，该命令显示一个日历。

显示当前月份的日历时，可以输入：

```
cal
```

显示某一年的日历时，需要指定年度。例如：

```
cal 1952
```

当指定年度时，确定键入了全部 4 个数字。例如，如果输入的是 **cal 52**，那么您将得到公元 52 年的日历。如果希望得到的是 1952 年的日历，需要使用 **cal 1952**。年度位于 1~9999 之间。

提示

当显示一年的完整日历时，输出会很长，屏幕无法完全显示。

如果担心日历的顶部在阅读之前滚动出了视野，那么您可以每次一屏地显示输出，即将命令的输出发送给 **less** 程序。例如：

```
cal 1952 | less
```

(我们将在第 21 章中讨论 **less** 程序。)

为了显示某个特定月份的日历，需要将这个月份指定为一个位于 1~12(1=一月)之间的数字，并且还要指定年度。例如，显示 1952 年 12 月的日历的命令如下：

```
cal 12 1952
```

得到的结果为：

```

December 1952
Su Mo Tu We Th Fr Sa
  1  2  3  4  5  6
  7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
    
```

如果希望指定一个具体的月份，那么必须同时指定月份和年度。例如，如果希望得到2009年7月的日历，则必须输入：

```
cal 7 2009
```

如果输入的是：

```
cal 7
```

那么得到的将是公元7年的日历。

如果不希望显示日期，只希望得到这一天是这一年中的第几天，即从1~365编号(1月1日=1，1月2日=2，等等)，那么也可以使用 `cal` 程序。只需在 `cal` 名称之后键入 `-j` 即可。下面举一个例子。

```
cal -j 12 2009
```

输出为：

```

      December 2009
Sun Mon Tue Wed Thu Fri Sat
      335 336 337 338 339
340 341 342 343 344 345 346
347 348 349 350 351 352 353
354 355 356 357 358 359 360
361 362 363 364 365
```

假定您希望推断某年是不是闰年。非 Unix 人士将查看该年度是否可以被4整除，如果可以，那么该年度就是闰年*。但是，作为一名 Unix 人士，您还有另外一种办法。您所需要做的全部就是显示该年度12月份的日期编号。如果12月以第366天结束，那么该年度就是闰年。否则，这个年度就不是闰年。试试下述命令，看看有什么结果：

```
cal -j 12 2000
```

```
cal -j 12 2008
```

```
cal -j 12 1776
```

顺便说一下，`-j` 代表儒略历(Julian)，即每年有365天或者366天的现代日历的名称。更多的信息，请参见下面方框中的内容。

儒略历和格里历(Gregorian calendar)

正常年度应该有365天，而闰年应该有366天，这一思想起源于儒略历，儒略历由儒略·恺撒于公元前46年制定*。

但是准确地说，一年平均有多长呢？如果4年中有3年是365天，第4年是366天，那么一年平均有365.25天。但是，自然界并不总是那么协作，这个值要比真实值长11分10秒。在16世纪的时候，这个小误差已经积累到大约10天，这意味着大家使用的日历没有与太阳和星星的移动相匹配。

* 唯一的例外就是世纪年，如果它们能够被400整除才是闰年。例如，2000年是闰年，而900年不是闰年。

* 实际上，该日历是一名研究生开发的，只是恺撒在论文上添加了自己的名字。

为了解决这个问题，罗马教皇格列高里(Gregory)十三世于 1582 年颁布法令，倡导世界应该修改日历，从而使所有的世纪年(如 1600 年、1700 年、1800 年等)并不全都是闰年。只有那些可以被 400 整除的世纪年(如 1600 年或者 2000 年)才是闰年。这一方案被称为格里历或者西洋新历(New Style Calendar)。为了校准当前的日期，格列高里进一步颁布法令，使 10 天神秘地消失了。

如果我们希望特别准确，那么我们可以说现代(格里)日历基于 400 年一循环，平均每年有 365.2425 天(一个有趣的事实是，在这个日历下，十亿分之一世纪(nanocentury)近似为 π 秒)。

罗马教皇于 1582 年颁布了法令。到 1587 年时，天主教欧洲国家已经实现了修改。而新教国家推迟了一些年才开始实施，最后一个国家是英国，直到 1752 年才进行修改。

随着英国的修改，美洲大陆也开始进行修改，但是，这个时候，误差已经增加至 11 天。因此，如果输入命令：

```
cal 9 1752
```

那么您将看到 1752 年 9 月 2 日和 9 月 14 日之间有一个缺口，这个缺口有 11 天。这一缺口对于太阳和星星的正常运转是必须的(至少对于英国和美国来说)。

8.5 Unix 提醒服务：calendar

我们刚刚讨论的 **cal** 程序显示日历。Unix 确实有一个名为 **calendar** 的命令，但是它与 **cal** 命令完全不同。**calendar** 程序基于一个自己创建的包含重要日期和消息的文件提供提醒服务。

对于该程序来说，您只须生成一个命名为 **calendar** 的文件。**calendar** 程序将在当前目录中查找这个文件(我们将在第 24 章中讨论当前目录)。这个文件中包含若干行文本，每行文本都包含一个日期，后面是一个制表符，再后面是一个提醒注释。例如：

```
October 21<Tab>Tammy's birthday
November 20<Tab>Alex's birthday
December 3<Tab>Linda's birthday
December 21<Tab>Harley's birthday
```

因为制表符是不可见的，所以在查看这个文件时看到的内容将如下所示：

```
October 21 Tammy's birthday
November 20 Alex's birthday
December 3 Linda's birthday
December 21 Harley's birthday*
```

一旦建立好 **calendar** 文件，就可以在任何时候输入 **calendar** 命令：

```
calendar
```

* 直到 1 月底才收到礼物。

当这样做时, **calendar** 程序将检查 **calendar** 文件, 并显示所有拥有今天或明天日期的文本行。如果“今天”是星期五, 那么 **calendar** 将显示接下来三天的提醒。

这里的要求是您要时不时地向 **calendar** 文件中添加一些文本行, 直到您的生活完全条理化了。当然, 在创建这样一个文件之前, 必须知道如何使用文本编辑器程序。最重要的文本编辑器是 **vi**(第 22 章)和 **Emacs**。

如果希望在每次登录时都自动运行 **calendar** 命令, 则可以将 **calendar** 命令放在 **.profile** 文件(Bash 或 Korn Shell)或者 **.login** 文件(C-Shell)中。详细细节请参见有关 shell 的各章讨论的内容。

一些 Unix 系统提供有内置的 **calendar** 文件, 该文件中包含一些有趣的条目。因此, 即使不想创建自己的 **calendar** 文件, 也可以时不时地运行 **calendar** 命令, 只是为了看看有什么事即将发生。您现在就可以试一试。

有关 **calendar** 命令和 **calendar** 信息文件格式的更多信息, 可以查看 Unix 联机手册(参见第 9 章)中有关 **calendar** 的条目。命令如下:

```
man calendar
```

8.6 查看系统信息: uptime、hostname、uname

Unix 系统中提供了几条命令可以用来查看系统的信息。首先, 我们讨论 **uptime** 命令。该命令用来显示系统已经运行多长时间(也就是连续运行)的有关信息:

```
uptime
```

下面是一些典型的输出:

```
11:10AM up 103 days, 6:13, 3 users,
load averages: 1.90, 1.49, 1.38
```

在这个例子中, 系统已经运行了 103 天 6 小时 13 分钟, 而且当前有 3 个用户标识登录。最后 3 个数字展现了一直等待执行的程序的数量, 分别是之前 1 分钟、5 分钟和 15 分钟的平均数。这些数字能够表示系统的负载。负载越高, 系统所做的工作就越多。

hostname 命令用来查看计算机的名称。如果经常登录不止一台计算机, 那么该命令非常管用。如果忘记了正在使用的是哪一个系统, 则可以输入:

```
hostname
```

uname 命令显示操作系统的名称。例如, 输入下述命令:

```
uname
```

将得到如下结果:

```
Linux
```

如果要得到操作系统更多的信息, 可以使用 **-a** 选项(all information, 全部信息):

```
uname -a
```

下面是一些示例输出：

```
Linux nipper.harley.com 2.6.24-3.358 #1
Mon Nov 9 09:04:50 EDT 2008 i686 i686 i386 GNU/Linux
```

这里最重要的信息就是我们使用的是 Linux 内核，且该内核的版本号是 2.6.24-3.358。

8.7 显示自己的信息：whoami、quota

whoami 命令显示您登录使用的用户标识。如果您有许多用户标识，而且忘记了正在使用的是哪个用户标识，那么使用这条命令非常便利。同理，如果您碰到了一个还登录着的计算机，那么 **whoami** 命令将显示当前的用户标识。只需输入：

```
whoami
```

如果您突然得了健忘症，忘记了自己的名字，那么 **whoami** 命令也特别有用。看看您的用户标识或许会给您提供线索。

如果您的系统上没有 **whoami** 命令，则可以尝试输入下面 3 个单独的单词：

```
who am i
```

最后一条显示您自己信息的命令是 **quota**。在共享系统上，系统管理员有时候会强加限制，例如，限制每个用户只允许使用多少磁盘空间。为了查看自己的限制，可以输入：

```
quota
```

注意，Unix 以 KB 或者千字节计算磁盘空间，其中 1KB=1024 字节(字符)。

8.8 显示其他用户的信息：users、who、w

以前，Unix 计算机是共享的，大多数时间，系统有多个用户同时登录。实际上，特大型的 Unix 计算机能够支持数十个(甚至几百个)用户同时使用。为了查看当前有哪些用户登录，Unix 系统提供了几条命令。现在，如果您与其他人共享系统，那么这些命令依然有用。

最简单的一条命令就是 **users**。只需输入命令本身即可：

```
users
```

该命令运行之后，将显示当前登录系统的所有用户标识，例如：

```
alex casey harley root tammy
```

第 4 章中讲过，在 Unix 系统中，只有用户标识拥有真正的身份。真正的人——也就是用户——由他们的用户标识表示。因此，您看不到真实人物的名字，看到的只是用户标识。

如果使用的是共享系统，那么 **users** 命令就非常有用。我可以告诉您，以我的经验来看，能够意识到其他人正在和您使用同一个系统会带来一种快乐的感觉，特别是当您深夜在办公室或者终端室工作时。知道其他地方还有其他人和您使用同一个系统，您会觉得自己和其他人连在一起。然而，许多人没有这种感觉，因为他们只体验到每个人都在使用自己的计算机。他们不知道共享一个系统的感觉。

您或许会问，如果在自己的计算机上使用 Unix，那么 **users** 命令还有使用意义吗？大多数时候答案是否定的：您可能会失望地看到自己是唯一登录系统的人。但是，当使用多个终端窗口或者虚拟控制台时(参见第 6 章)，如果您在每个窗口或控制台分别登录一次，那么在运行 **users** 命令时，您的用户标识将显示不止一次。同理，如果您使用几个不同的用户标识从虚拟控制台登录，那么在运行 **users** 命令时，所有的用户标识都将显示出来。因此，起码在您觉得孤独时，可以假装系统中还有其他人。

下一条命令是我们在第 4 章中已经讨论过的 **who** 命令。这条命令要比 **users** 命令显示更多的信息。对于每个用户标识，**who** 命令将显示终端的名称、用户标识的登录时间，而且如果合适的话，还会显示用户标识使用哪台远程计算机连接到系统上。

下面是在一个人们通过远程连接登录的系统中运行 **who** 命令的输出：

```
tammy  tty1    Nov 10 21:25
root   tty2    Nov 11 15:12
casey  pts/0    Nov 11 10:07 (luna)
harley pts/1    Nov 11 10:52 (nipper.harley.com)
alex   pts/2    Nov 11 14:39 (thing.taylored-soft.com)
```

在这个例子中，用户标识 **tammy** 使用终端 **tty1** 登录，**root**(超级用户)在终端 **tty2** 上登录。这两个终端是主计算机的虚拟控制台。正如第 7 章所述，名称 **tty** 通常用作“terminal(终端)”的缩写。在这个例子中，碰巧用户 **tammy** 是系统管理员，而且她当前登录了两次：一次使用她个人的用户标识，一次作为超级用户。

下面我们继续看另一个用户标识 **casey**，**casey** 从一台名为 **luna** 的计算机上登录，这台计算机位于局域网中。最后是另外两个用户标识 **harley** 和 **alex**，他们通过网络远程登录系统。

如果希望查看系统上用户标识更多的相关信息，那么您可以使用 **w** 命令。**w** 名称的含义是“Who is doing what? (谁正在做什么)”下面是一些输出样本：

```
8:44pm up 9 days, 7:02, 3 users, load average: 0.11, 0.02, 0.00
USER  TTY      FROM LOGIN@  IDLE   JCPU   PCPU   WHAT
tammy  console -    Wed9am 2days 2:38   0.00s  -bash
harley pts/1    -    12:21   0:00   0.01s  w
alex   tty0     luna 13:11 20:18 43.26s 1.52s  vi birdlist
```

输出的第一部分显示系统统计信息，这一部分信息已经在本章前面讨论 **uptime** 命令时看到过。在这个例子中，系统已经运行了 9 天 7 小时 2 分钟，当前有 3 个用户标识登录。最后 3 个数字显示的是一直等待执行的程序的数量，分别是之前 1 分钟、5 分钟和 15 分钟的平均数。这些数字可以使您知道系统的负载。负载越高，系统所做的工作就越多。

在第一行下面，是 8 列信息，分别解释如下。

USER: 当前登录系统的用户标识。在这个例子中，他们分别是 **tammy**、**harley** 和 **alex**。

TTY: 各个用户标识使用的终端的名称。

FROM: 用户标识登录系统所使用的远程计算机的名称。在我们的例子中，**tammy** 和 **harley** 直接从主机计算机上登录。但是 **alex** 从另一台名叫 **luna** 的计算机上登录。

LOGIN@: 用户标识的登录时间。

IDLE: 用户上一次按键后已经过去的时间。这段时间称为空闲时间。在这个例子中，**tammy** 已经空闲了大约 2 天，**alex** 已经空闲了 20 分钟 18 秒。如果您在等待一个耗时的任务结束，而且在等待过程中没有键入信息，那么 **w** 命令将显示您空闲。

JCPU: 自登录后所有进程总共使用的处理器时间。其中“J”代表“jobs(作业)”。

PCPU: 当前进程所使用的处理器时间。其中“P”代表“process(进程)”。处理器时间或者用秒(如 **20s**)或者用分钟和秒(如 **2:16**)表示。在计算机价格昂贵，而且处理器时间是一个贵重商品的时代，这些数字非常有价值。

WHAT: 当前正在运行的命令。在我们的例子中，**tammy** 正在运行 **Bash shell**；**alex** 正在使用 **vi** 编辑器(参见第 22 章)编辑一个名为 **birdlist** 的文件；而 **harley** 正在运行 **w** 命令。

将所有这些结合在一起，我们可以推断出 **tammy** 在控制台登录，使用了一个 **shell** 提示，但是已经有两天没做任何事情了。或许这就是系统管理员，喜欢一直登录着。我们还看到 **alex** 正在编辑一个文件，但是已经有 20 分钟没做任何事情了。或许他正在休息。

注意，无论何时，当您运行 **w** 命令时，都将看到自己正在运行 **w** 命令。在我们的例子中，**harley** 就是这种情况。

默认情况下，**w** 命令显示所有登录到系统中的用户标识的信息。如果只需要得到一个用户标识的信息，那么可以在 **w** 命令之后输入该用户标识的名称。例如，假设您刚刚登录系统，并且输入 **users** 命令查看了还有哪些用户登录到系统。您看到：

```
alex casey harley tammy weedly
```

如果您希望查看 **weedly** 正在做什么，可以输入：

```
w weedly
```

名称含义

CPU

在大型计算机时代，计算机的“大脑”(现在称之为处理器)非常庞大，需要一个大盒子才可以装下，我们称它为中央处理单元或者 CPU。

现在，即便大多数人已经不再使用大型计算机，但是我们仍然使用术语 CPU 作为“处理器”的同义词。

8.9 终端临时上锁：lock

正如第 4 章所述，在登录状态下离开计算机是很不好的。这样有些人可能会以您的用

户标识的名义输入命令，从而导致不少麻烦。例如，恶意的人可以删除您的所有文件，或者用您的名字向系统管理员发送不礼貌的电子邮件等。

但是，如果您只是需要离开终端一小会，那么注销以后再登录也很麻烦。如果您使用了一个或者多个终端窗口登录远程主机，或者已经建立好自己喜欢的整个工作环境，那么这时注销尤为麻烦。

这种情况下，可以使用 **lock** 命令。该命令告诉 Unix 您希望临时锁住终端。除非您输入一个特殊的口令，否则终端将一直保持锁住状态。使用该命令只需输入：

lock

Unix 将显示：

Key:

输入用于终端解锁的口令。该口令可以根据自己的愿望设定，它与登录口令没有任何关系。实际中，最好不要使用登录口令作为该口令。在输入口令时，Unix 不回显口令，以防被别人看见。在输入口令之后，Unix 将显示：

Again:

这是要求再次键入口令，以确保口令输入过程中没有出错。

一旦您输入并再次输入这个特殊口令后，Unix 将冻结终端。除非输入口令(不要忘记按<Return>键)，否则不管别人在终端上键入什么内容，终端都不会有任何反应。一旦输入正确的口令，Unix 将重新激活终端，您又可以继续工作。

提示

如果登录了一个或者多个远程主机，那么您应该分别锁住每个会话。

如果您工作的环境必须共享计算机而且还有人等着用，那么离开一段不是很短的时间(例如，去吃饭)而将终端锁上就不太好。因为 Unix 是在一个基于共享的环境中开发的，所以 **lock** 命令有一个内置的限制：在一个特定的时间量之后终端会自动解锁。

默认情况下，**lock** 将冻结终端 15 分钟。但是，如果希望替换这个默认值，那么一些版本的 **lock** 允许在输入命令时指定另外一个时间限制。在该命令之后，留一个空格，然后键入-(一个连字符)，后面再跟一个数字。例如，将终端锁住 5 分钟的命令为：

lock -5

您可能会问，如果有人将终端锁上，然后离开后不回来了怎么办？最终，如果时间到了，命令会自动解锁。如果在上锁时间耗尽之前，需要激活终端，那么系统管理员可以输入 **root**(超级用户)口令。**lock** 总是可以接受 **root** 口令，后者在某种程度上就像是万能钥匙一样。

请记住，如果您将终端锁上了，但是离开后不再回来，那么最终会有人过来，发现您的终端已被重新激活，从而以您的用户标识登录。无论他们用您的用户标识导致什么问题，责任都在您身上。

8.10 请求 Unix 提醒何时离开: leave

在计算机上工作时经常需要全神贯注, 很容易忘记时间。为了帮助您及时完成其他事情, 只需输入下述命令:

```
leave
```

顾名思义, 您可以使用 **leave** 命令提醒自己何时到时间, 应该离开了。您还可以使用该命令提醒什么时候该吃饭了。例如, 如果您喜欢每过一会(假如 20 分钟)站起来伸一下腰, 那么您可以请求一个提醒。

当输入该命令时, **leave** 将询问您一个时间:

```
When do you have to leave?
```

以 *hhmm*(先是小时, 后是分钟)的格式输入您希望离开的时间。例如, 如果希望在 10:33 离开, 可以输入 **1033**。

时间输入既可以是 12 小时制也可以是 24 小时制。例如, **1344** 意味着 1:44 PM(下午)。而如果输入的小时数小于或者等于 12, 那么 **leave** 假定该时间位于接下来 12 小时之内。例如, 如果现在时间是 8:00 PM, 而您输入的是 **855**, 那么 **leave** 将把它解释为 8:55 PM, 而不是 8:55 AM。

另一种输入 **leave** 命令的方法就是在命令行上立即输入时间。在命令之后, 留下一个空格, 然后键入时间。例如, 要在 10:30 离开, 可以输入:

```
leave 1030
```

如果需要在特定的时间间隔之后离开, 则可以输入一个+(加号), 后面跟着分钟数。例如, 如果需要在 15 分钟之后离开, 可以输入:

```
leave +15
```

确保不要在+字符之后留空格。

提示

当注销系统时, Unix 会废除悬挂的 **leave** 命令。因此, 如果使用了 **leave** 命令, 但是随后注销了系统并再次登录, 那么您需要重新运行 **leave** 程序。

一旦输入了 **leave** 命令, Unix 就会定期检查还剩下多长时间。在指定的时间前 5 分钟, Unix 会显示:

```
You have to leave in 5 minutes.
```

当还剩下 1 分钟时间时, 您将会看到:

```
Just one more minute!
```

当时间到时, Unix 显示:

Time to leave!

从这时起, Unix 将不停地提醒您, 每分钟一次, 直到您注销系统:

You're going to be late!

最后, 在这样提醒 10 次之后, 您将会看到:

You're going to be late!

That was the last time I'll tell you. Bye.

或许该程序应该命名为 **mother**。

提示

在登录时, 可以自动地运行 **leave** 命令, 为此, 只需将该命令放在您的初始化文件中即可(对于 Bash 或 Korn Shell 来说是 **.profile**, 对于 C-Shell 来说是 **.login**)。

这意味着每次登录时, 系统都会向您询问希望工作多长时间。通过这种方式, 您不用再看着时钟了, Unix 会为您做好此事。

如果以这种方式使用 **leave**, 但是在当前会话中不希望 **leave** 运行, 那么您可以在第一个提示处按下 <Return> 键, 该程序就会终止。

提示

当您需要一会之后就去做某事, 而且需要一个提醒时, 使用 **leave** 程序非常便利。例如, 假如您正在工作, 且希望每隔 15~20 分钟站起来活动一下。但是, 您全身心地投入到工作中, 很难记起要休息一会儿。为此只需简单地输入下述命令:

```
leave +20
```

15 分钟之后(提前 5 分钟), **leave** 将发出一个警告提醒您注意。在接下来的 5 分钟时间内, **leave** 将显示更多警告。

在休息之后, 可以再次输入这条命令, 15 分钟之后您将会得到另一个提醒*。

8.11 内置计算器: **bc**

Unix 最有用(但受到最少欣赏)的程序之一就是 **bc**, 它是一个功能齐全、可编程的科学计算器。许多人不乐意学习如何使用 **bc**。“我讨厌 **bc**,” 他们轻蔑地说, “没有人愿意使用

* 著名的医生 Janet G. Travell(1901-1997), 她开办了一家针对美国总统 John Kennedy 和 Lyndon Johnson 的私人诊所, 她是骨骼肌疼痛和机能障碍以及慢性疼痛方面的专家。

在编写她的经典著作 *Myofascial Pain and Dysfunction: The Trigger Point Manual* 时, Travell 医生发现为了维持她的舒适和健康, 她每隔 15~20 分钟需要一个短暂的休息。在休息期间, 她将站起来、来回走动并伸伸懒腰。

您将会发现, 在工作过程中, 进行这样的休息将使您十分舒适, 特别在您忍受头痛、背痛以及眼睛疲劳时。只需使用命令 **leave +20** 提醒您何时要活动一会、伸伸懒腰。一旦您开始这种练习, 您将在数小时之内感觉到舒服(这是 Unix 有利于健康的另一个例子)。

它。”不要犯这种错误，一旦学会了使用 **bc**，您就会发现它是做快速计算的无价之宝。

如果使用的是桌面环境(第 5 章)，那么您极有可能发现一些基于 GUI 的计算器程序可供使用。这些程序看上去十分漂亮——它们实际上是在计算机屏幕上绘制了一个计算器图像，但是，对于一步一步的工作或者大量的计算来说，**bc** 要更好用些。此外，**bc** 是基于文本的程序，这意味着可以在任何终端上的命令行中使用它。

为了解释 **bc**，这里首先进行一个简短的技术小结。如果您不了解所有的数学和计算机术语，不用担心。在接下来的几节中，将通过几个例子解释如何使用 **bc** 做基本的运算(很容易的)。

技术小结：**bc** 是一个完全可编程的数学解释器，它提供扩展精度。每个数字可以按所需要的数字位存储，而且小数点右边可以达到 100 位。可以对从 2 到 16 的各种进制的数进行操作，而且还可以方便地从一种进制转换到另一种进制。

使用 **bc** 时，既可以借助键盘输入进行运算(这种运算会被立即处理)，也可以运行以文件形式存储的程序。**bc** 的编程语法与 C 语言相似。**bc** 中可以定义函数并使用递归。**bc** 中还有数组、局部变量和全局变量。在 **bc** 中还可以编写自己的函数并存储成文件，然后 **bc** 可以加载它们并自动解释。

bc 自带了一个函数库，提供有下述函数：**sin**、**cos**、**arctan**、**ln**、指数和贝塞尔函数。

要了解更多的信息，可以使用下述命令显示联机手册中有关 **bc** 命令的描述：

```
man bc
```

(我们将在第 9 章中讨论 Unix 联机手册。)

8.12 使用 **bc** 进行计算

大多数时候，使用 **bc** 进行的都是常规计算，这比较简单。启动 **bc** 程序时，需要输入命令：

```
bc
```

如果您希望使用 **bc** 内置的数学函数库(参见下面)，那么在启动程序时需要使用 **-l**(library, 库)选项：

```
bc -l
```

bc 一旦启动，就没有具体的提示了。可以一个接一个地输入算式，每按下<Return>键，**bc** 就计算刚才键入的算式，并显示答案。例如，如果输入了：

```
122152 + 70867 + 122190
```

那么 **bc** 将显示：

```
315209
```

现在可以输入一个新算式了。如果希望在同一行上输入不止一个算式，那么各个算式

之间需要以分号隔开*。bc 将把每个算式的结果分别显示在一个单独的行上。例如，如果输入：

```
10+10; 20+20
```

那么将会显示：

```
20
```

```
40
```

当不再使用 bc 时，可以通过告诉它再没有数据了而结束程序。这一操作通过按[^]D(eof 键，参见第 7 章)实现。另外，也可以输入 quit 命令。

图 8-1 示范了 bc 中可用的基本运算。其中加法、减法、乘法、除法和平方根都直接明了，和平时一样。取模运算是除法运算的余数。例如， $53\%10$ 等于 3。指数运算是对一个数乘方。例如， 3^2 的意思是“3 的 2 次幂”，结果为 9。乘方必须是一个整数，但是可以是负数。如果用负数乘方，则要用括号将负数括起来，如 $3^{(-1)}$ 。

运算符	含义
+	加法
-	减法
*	乘法
/	除法
%	取模
^	指数
sqrt(x)	平方根

图 8-1 bc：基本运算

bc 遵循一般的代数运算规则：乘法、除法及取模的优先级高于加法和减法，指数的优先级最高。和代数运算一样，可以使用括号改变运算的顺序。因此， $1+2*3$ 等于 7，而 $(1+2)*3$ 等于 9。

除了基本的运算之外，bc 函数库中还有几个有用的函数。这些函数如图 8-2 所示。

函数	含义
s(x)	x 的正弦；其中 x 的单位是弧度
c(x)	x 的余弦；其中 x 的单位是弧度
a(x)	x 的反正切；其中 x 的单位是弧度
ln(x)	x 的自然对数
j(n, x)	x 的 n 次整阶贝塞尔函数

图 8-2 bc：数学函数

* 第 10 章中将会讨论，当位于 shell 提示时，可以在同一行上键入不止一条命令，各条命令之间用分号隔开。

如果需要使用库中的函数，则需要使用下述命令启动 **bc**：

```
bc -l
```

当使用这个命令时，**bc** 自动地将标度因子(scale factor)设置为 **20**(参见下面)。

正如前面所述，**bc** 可以进行任意精度的计算。也就是说，它可以根据需要，使用任意的位数进行运算。例如，您可以要求它将两个 100 位的数字相加(我测试过)。

但是，默认情况下，**bc** 假定做整数运算。也就是说，**bc** 将忽略小数点右边的数字。如果希望使用小数值，那么您需要设置一个标度因子，告诉 **bc** 您希望保留小数点后多少位。根据需要，您可以将 **scale** 设置为希望的标度因子值。

例如，要保留小数点后面 3 位，可以输入：

```
scale=3
```

从现在起，随后所有的运算都是 3 位小数运算，后面的数字都被舍去。

如果希望查看标度因子的值，只需简单地输入：

```
scale
```

bc 将显示 **scale** 的当前值。

在启动 **bc** 时，**scale** 的值被自动设置为 **0**。一个常见的错误就是开始计算时没有设置标度因子。例如，假设您刚启动了 **bc**，输入了：

```
150/60
```

bc 显示：

```
2
```

现在您再输入：

```
35/60
```

bc 显示：

```
0
```

最后，您弄清楚了问题的所在。您的结果被取舍了，因此您需要设置一个合适的标度因子：

```
scale=3
```

这时 **bc** 将显示您希望看到的结果(试试看)。

记住，当使用数学函数库时，**bc** 启动时自动将标度因子设置为 **20**。基于这一原因，许多人通常使用 **bc -l** 启动 **bc**，即使他们不使用数学函数库(我就一直这样做)。

8.13 在 **bc** 中使用变量

bc 不仅仅是一个计算器。它实际上还是一种功能完整的数学编程语言。跟所有的编程语言一样，**bc** 允许设置并使用变量。

变量由变量名和值组成。在 **bc** 中，变量名由一个小写的字母构成。也就是说，**bc** 中

共有 26 个变量，从 **a** 到 **z**(一定要确保不要使用大写字母，当使用基时才使用它们——参见下面)。

在设置变量的值时，使用=(等号)字符。例如，将变量 **x** 的值设置为 **100**，可以输入：

```
x=100
```

显示变量的值时，只需输入变量名。例如：

```
x
```

bc 将显示变量 **x** 的当前值。默认情况下，在变量没有赋值之前，**bc** 假定所有变量的值为 **0**。

变量的使用非常简单，并且还增强了 **bc** 的功能。下面举例说明它的基本原理。

Gaipajama 王公被 Unix 提供的功能深深打动。作为他尊重的表示，他决定为您提供您体重两倍的红宝石(每磅值 1000 美元)以及您体重 1/3 的钻石(每磅值 2000 美元)。很显然，Gaipajama 王公按批发价购买的宝石。

您的体重是 160 磅，那么王公的礼物值多少钱呢？为了解决这个问题，可以启动 **bc** 并输入：

```
w=160  
r=(w*2)*1000  
d=(w/3)*2000  
r+d
```

答案显示为：

```
426000
```

因此，送给您的礼物值 426 000 美元。

但是等一等，一旦王公意识到他的承诺代价太大时，他会说：“我说过送给您的礼物是以磅为单位计算的吗？我说的是以公斤计算。”

因为 1 公斤等于 2.2 磅，所以您可以很快地将 **w** 变量的值转换成公斤制：

```
w=w/2.2
```

现在再重新输入红宝石和钻石的值：

```
r=(w*2)*1000  
d=(w/3)*2000  
r+d
```

新的答案显示为：

```
192000
```

因此，由于采用公制单位，所以王公节省了 234 000 美元。同时，您向他演示了如何基于旧值为变量设置一个新值，在这个例子中，即 **w=w/2.2**。

* 有关 Gaipajama 王公的更多信息，请参见 Hergé 的 *Cigars of the Pharaoh*。

8.14 在 bc 中使用不同的基

一般情况下, **bc** 计算时使用 10 作为基(如果还不知道基是什么, 那么您可以跳过本节)。但是, 有时候您可能需要使用另一种基进行计算。例如, 在计算机科学中, 有时候需要基 16(十六进制)、基 8(八进制)或者基 2(二进制)。我们将在第 21 章中讨论这些计数制。

bc 允许对输入和输出指定不同的基。具体操作时, 需要设置两个特殊的变量: **ibase** 是用于输入的基; **obase** 是用于输出的基。

例如, 如果希望以基 16 显示答案, 可以输入:

```
obase=16
```

如果希望以基 8 输入数字, 则需要使用:

```
ibase=8
```

上一节中说过, 默认情况下, 在设置变量值之前, 变量的值为 0。但是 **ibase** 和 **obase** 例外: 这两个变量都被自动地设置为 10, 这样就可以使用基 10 进行计算。如果希望以另一种基进行计算, 则需要设置这两个变量的值, 值的范围为 2~16。

您还应该明白 **ibase** 和 **obase** 的值并不影响 **bc** 内部对数值的操作。它们唯一的效果就是指定输入和输出过程中数值如何转换。

对于大于或等于 10 的基值, **bc** 分别使用大写字母 **A**、**B**、**C**、**D**、**E** 和 **F** 表示值 10、11、12、13、14 和 15。记住一定要使用大写字母, 如果使用了小写字母, 那么 **bc** 将会认为它是变量, 从而使结果出错。

为了方便起见, 无论设置的输入基是什么, 都可以使用这些大写字母。例如, 即使您现在使用的基是 10, 表达式 **A+1** 的值也是 11。

和其他变量一样, 直接输入 **ibase** 和 **obase** 自身就可以获得它们的当前值:

```
ibase; obase
```

但是, 一定要小心。因为一旦设置了 **obase** 的值, 所有的输出都将以这个基进行显示, 显示的值可能会对您产生麻烦。例如, 如果您输入:

```
obase=16  
obase
```

那么您将看到:

```
10
```

这是因为此时所有的输出都以基 16 显示, 而在基 16 中, 值“16”就表示为 10。同理, 一旦修改了 **ibase**, 在输入时也必须特别小心。例如, 假设您设置了:

```
ibase=16
```

现在希望将 **obase** 设置为基 10, 因此您输入:

```
obase=10
```

但是，您忘了现在的输入是基 16，而 **10** 在基 16 中其实是“16”。因此，**obase** 还是被设置成基 16。

为了避免出现这样的错误，可以使用字母 **A** 到 **F**，无论 **ibase** 的值是多少，它们仍然是原来的值。因此，如果事情出现了混乱，您总可以这样重新设置基：

```
obase=A; ibase=A
```

下面举两个修改基的例子。在第一个例子中，希望将两个十六进制(基 16)的数 **F03E** 和 **3BAC** 加在一起。输入：

```
obase=16
ibase=16
F03E + 3BAC
```

bc 显示答案：

```
12BEA
```

在第二个例子中，希望将十六进制的数 **FFC1** 转换成二进制(基 2)。重新设置基：

```
obase=A; ibase=A
```

然后输入：

```
obase=2; ibase=16
FFC1
```

bc 显示答案：

```
1111111111000001
```

提示

bc 不仅仅是一个计算器程序。它还是一个复杂的数学编程系统，有自己内置的编程语言。在本章中，只解释了最基本的特征。如果您有时间，那么建议您研究一下 **bc**，看看它还能做些什么。

其中最好的方式就是阅读 **bc** 的联机手册。命令如下：

```
man bc
```

(我们将在第 9 章中讨论 Unix 联机手册。)

8.15 逆波兰表示法

最初，**bc** 程序基于一个叫 **dc**(desk calculator, 桌面计算器)的程序。**dc** 是最古老的 Unix 程序之一，甚至比 C 语言还要早。实际上，**dc** 的最初版本是使用编程语言 B(C 的祖先)于

1970 年编写的。一会之后，我们将进一步讨论 **bc** 和 **dc** 的关系。但是，现在我将讲授一些有关 **dc** 的内容，**dc** 本身就是一个十分有趣的程序，因为像 **bc** 一样，它是一个可以立即使用的程序。

我们从一个技术描述开始：**dc** 是一个交互式、不定精确度的计算器，它仿真了一个使用逆波兰表示法(Reverse Polish notation)的栈机器。

很明显，**dc** 并不是那种可以吸引所有人的程序类型：如果对数学或者计算机科学不感兴趣，那么您可以自由地选择跳过该讨论。但是，如果您倾向于技术，那么对 **dc** 的理解十分重要，原因有以下几方面。

首先，正如前面所述，**dc** 使用了所谓的逆波兰表示法。尽管这一思想现在对您来说可能没有任何意义，但是它是一个重要的概念，如果您将要学习数学、工程或者计算机科学，那么应该重视这一概念。

其次，为了学习 **dc**，您需要理解栈的思想(稍后解释)，这是一个对计算机科学家和程序员都很重要的概念。

最后，使用 **dc** 所需的思考方式与使用 Unix 所需的思考方式相同。因此，花一点时间学习 **dc**(如果乐意的话，可以自己学习如何使用它)将使您更进一步成为 Unix 人士。

我们将从逆波兰表示法的解释开始关于 **dc** 的讨论。在下一节中，将介绍栈的概念。一旦理解了这两个基本思想，您就能够通过联机文档自学如何使用 **dc**。

1920 年，一位叫 Jan Lukasiewicz(1878-1956)的波兰数学家观察我们书写算术表达式的方式，发现可以通过将运算符放在操作数之前，使表达式更加紧凑。采用这种方式，我们就能够不使用圆括号或者方括号书写复杂的表达式。下面示范一个简短的例子来描述这一思想。

假设您希望将 34 加上 25，然后把和乘以 15。使用标准的表示法时，写法如下所示：

$(34 + 25) * 15$

因为运算符(在这个例子中是+(加号)和*(乘号))位于操作数的中间，所以我们称这种表示法为中缀表示法(infix notation)。

Lukasiewicz 的系统使用前缀表示法(prefix notation)，在这种表示法中，先书写运算符，然后才是操作数。例如：

$* + 34 25 15$

为了求前缀表示法的值，我们从左向右，每次一个地处理元素。在这个例子中，首先是*运算符，它告诉我们只要获得两个数字就执行一个乘法运算。然后我们遇到+运算符，它告诉我们只要获得两个数字就执行一个加法运算。

接下来，连续获得两个数字 34 和 25，因此我们执行加法运算，得到一个和 59。记住这个 59，我们继续，然后遇到了数字 15。现在可以执行乘法 $59*15$ ，获得最后结果 885。

为了纪念 Lukasiewicz(一名著名的数学家、逻辑学家和哲学家)，通常将前缀表示法称

为波兰表示法^{*}(Polish notation)。对于计算机科学家来说,波兰表示法十分重要,因为它紧凑、直接,而且求值过程很高效。

1957年,奥地利计算机科学家 Charles Hamblin 撰写了两篇论文,在这两篇论文中,他提议在基于栈的计算系统中使用波兰表示法的一种变体(我们将在下一节中讨论栈)。他描述的变体就是将运算符放在操作数之后,这种表示法称为后缀表示法(postfix notation)。

为了描述后缀表示法,我们重新考虑前面的表达式。在后缀表示法中它如下所示:

34 25 + 15 *

为了求这种表达式的值,我们从左向右处理元素。首先,看到两个数字 **34** 和 **25**,这两个数字我们必须记住。接下来,我们看到 $+$ 运算符,这告诉我们将之前两个可用的数字加起来。在这个例子中,将 **34** 加上 **25** 得到 **59**,这个数字我们也必须记住。

接下来,我们看到数字 **15**,这个数字也要记住。最后,看到 $*$ 运算符,它告诉我们将最后两个数字乘在一起。在这个例子中,两个数字是 **59** 和 **15**,将它们乘起来获得最后结果 **885**。

后缀表示法特别适合自动计算,这是因为表达式可以通过记住数字并应用遇到的运算符,以一种直接的方式从左向右求值。对于中缀表示法,括号和其他类型的优先级(例如乘法必须先于加法完成)通常要求某些运算必须一直等到其他运算完成时才能进行。而后缀表示法就没有这种问题。

为了纪念 Lukasiewicz, 后缀表示法通常称为逆波兰表示法或者简称为 RPN。多年以来,波兰表示法和逆波兰表示法在许多计算机系统中都得到广泛的应用。例如, Lisp 编程语言和 Tcl 脚本语言中就使用了波兰(前缀)表示法。Forth 编程语言和 PostScript 页面描述语言就使用了逆波兰(后缀)表示法。

或许 RPN 最有名的应用就是作为 HP 计算器的基础。HP 计算器已被广大科学家和工程师使用了许多年。第一个此类计算器是 HP 9100, 于 1968 年发明。

从那时起, RPN 计算器就开始流行,因为一旦您理解了 RPN, 相对于传统的中缀表示法,它要快许多,并且使用更容易。例如,当您使用 RPN 计算器时,每个计算的结果会立即显示。这意味着在输入运算时可以看到局部结果,从而使错误的捕捉相对容易。而使用传统中缀表示法的计算器,情况就不是这样了。如果您输入了一个使用标准优先级规则的表达式,那么除非整个运算结束,否则结果是不会显示出来的。

1970年,贝尔实验室的一名研究人员 Robert Morris, 受 HP 计算器的启发,使用 RPN 开发了一个基于 Unix 的交互式计算器程序。Morris 将这个程序称为 **dc**(desk calculator, 桌面计算器)。**dc** 是一个神奇的工具,但是它要求用户学习如何使用 RPN。

几年之后, Morris 和另一名研究人员 Lorinda Cherry 一起编写了另一个名叫 **bc** 的程序,该程序允许用户使用更传统的中缀表示法书写算式。**bc** 的工作方式是将输入转换成 RPN, 然后调用 **dc** 完成实际工作。换句话说, **bc** 是 **dc** 的“前端”。这允许人们选择他们喜爱的系统: **dc** 的后缀表示法或者 **bc** 的中缀表示法。

^{*} Jan Lukasiewicz 于 1878 年 12 月 21 日出生于 Galicia(奥地利最大和最北部的省)的 Lwow 市。Galicia 是 1772 年第一次瓜分波兰(First Partition)的产物。尽管按照正式说法, Lukasiewicz 是奥地利人,但是按照民族来说,他是波兰人,而且那时 Lwow 也由波兰人统治。现在, Lwow 被称为 Lviv, 成为西乌克兰地区最大的城市。

下面是一件有趣的事情:我的爷爷 Irving Hahn(1895-1986)年轻时就生活在 Lwow, 在那里,他学习过理发。此外,我的生日也是 12 月 21 日。

若干年之后，作为 GNU 项目(参见第 2 章)的一部分，**bc** 作为一个独立的程序完全进行了重写。因为许多现代的 Unix(包括 Linux 和 FreeBSD)都使用 GNU 实用工具，所以当您现在使用 **bc** 时，使用的是一个不再依赖于 **dc** 的独立程序。当然，**dc** 仍然可以单独使用。

8.16 基于栈的计算器：dc

考虑下述 RPN(后缀)表示法的例子：

```
34 25 + 15 *
```

dc 以上一节中描述的方式求这个表达式的值，从左向右，每次读取一个元素。**dc** 每遇到一个数字，这个数字的值将被记住。**dc** 每遇到一个运算符，都必须执行合适的运算，而结果必须被记住。

这就出现了一个问题，**dc** 如何记录各个必须记住的元素呢？答案就是使用所谓的栈。

在计算机科学中，有各种不同的数据结构可以用来存放数据。每种类型的数据结构能够根据自己的一组明确规则存储和检索数据。其中最常见的数据结构有列表、链表、关联数组、哈希表、栈、队列、双头队列(双端队列)以及多种基于树的结构。在本节中，我们将集中讨论栈，因为它就是 **dc** 程序使用的数据结构。

栈是一种根据“后进先出(last in first out, LIFO)”过程每次存储或者检索一个数据元素的数据结构。下面是栈的工作方式。

刚开始时栈是空的。为了在栈中存储一个数据元素，需要将数据元素压入(push)栈中。现在数据就位于栈的顶部。每次一个，您可以在栈中压入任意多的数据元素。每压入一个元素，栈中的所有元素都向下移一级。因此，在任何时候，栈的顶部都包含刚刚压入到栈中的数据。从栈中检索数据时，只需从栈的顶部取数据即可。当这样做时，我们称之为栈的弹出(pop)运算。

换句话说，当弹出栈时，会检索压入到栈中的最后一个值。这就是为什么将栈描述为 LIFO(后进先出)的原因。

下面举一个栈的具体例子，想象一下自助餐厅中的弹簧支承的盘子叠。盘子一个一个地压入到“栈”中。当需要盘子时，必须将栈顶部的那一个盘子弹出。您不能获取其他的盘子。如果基于某些原因，需要最底部的那个盘子，那么您不得不将该盘子顶上的所有盘子逐个弹出。

dc 程序使用栈以这种方式解释 RPN 表示的算术表达式。在这样做时，**dc** 遵循一个简单的过程：从左向右读取表达式，每次一个元素。如果遇到的是数值，则将它压入到栈中。如果遇到的是运算符，则从栈中弹出合适数量的元素，执行运算，然后将结果压入到栈中。

为了描述该运算过程，我们重新考虑前面的例子：

```
34 25 + 15 *
```

下面是 **dc** 解释这个表达式的步骤：

- (1) 读取值 **34** 并将它压入到栈中。
栈中包含有: **34**
- (2) 读取值 **25** 并将它压入到栈中。
栈中包含有: **25 34**
- (3) 读取+(加号)。为了执行加法, 需要两个值, 因此……
- (4) 从栈中弹出 **25** 和 **34**, 并将它们相加。将结果(**59**)压入到栈中。
栈中包含有: **59**
- (5) 读取值 **15** 并将它压入到栈中。
栈中包含有: **15 59**
- (6) 读取*(乘号)。为了执行乘法, 需要两个值, 因此……
- (7) 从栈中弹出 **15** 和 **59**, 并将它们相乘。将结果(**885**)压入到栈中。
栈中包含有: **885**

如果还不能立即掌握 RPN, 那么您不用担心。通过多练习 **dc** 程序, 您一定能够体会出来。实际上, 真正理解 RPN 的最好方法就是多多练习。

在我们的例子中, 示范了每个步骤上栈中的内容。而在 **dc** 中, 您看不到栈。但是, 在任何时候都可以使用 **P**(**print**, 打印)命令*显示栈的顶部。为了示范该命令的工作方式, 我们启动 **dc** 程序, 然后输入下述两行(不要忘记了在每行的末尾按<Return>键)。

```
34 25 + 15 *
P
```

在输入了第一行之后, **dc** 程序执行运算。但是, 您看不到任何东西。一旦您输入了第二行(**p** 命令), **dc** 将显示栈顶元素的值, 在这个例子中是 **885**, 这是上一个运算的结果。

如果希望看到整个栈, 可以使用 **f** 命令:

```
f
```

为什么 **dc** 不能在每次输入一个新行时自动显示栈顶元素的值呢? 答案是如果 **dc** 在每输入一行时显示一些东西, 那么它将导致屏幕混乱。实际上, **dc**(像大多数 Unix 程序一样)尽可能不显示内容。需要时, 您自己可以查看栈顶元素的值。

为了帮助您开始使用 **dc**, 图 8-3 概括了一些最重要的 **dc** 命令。除了该列表之外, 自己学习使用 **dc** 的最好方法就是阅读联机手册(参见第 9 章)并多加练习。查看 **dc** 手册页面的命令是:

```
man dc
```

dc 中小数点后面的数字位数称为“精确度”。精确度默认值是 **0** 位。为了修改小数点的位数, 需要将位数压入到栈中, 然后输入 **k** 命令。这个值可以任意大。例如, 将精确度修改为 **14** 位数字, 可以使用:

```
14 k
```

* 正如第 7 章中解释的, 因为以前的 Unix 终端将它们的输出打印在纸上, 所以术语“**print**(打印)”通常用作“**display**(显示)”的同义词。因此, **dc** 的打印命令显示栈顶的值。许多 Unix 程序使用这一传统。

如果要显示当前的精确度，可以使用 **K**(大写“K”)命令——这将把当前精确度压入到栈顶，然后再使用 **p** 命令显示实际值：

K p

最后，在停止 **dc** 时，或者通过按下 **^D** 表示已经没有数据了，或者使用 **q**(quit, 退出)命令。

命令	含义
q	退出
p	显示栈顶
n	弹出栈并显示值
f	显示栈的全部内容
c	清除(清空)栈
d	复制栈顶值
r	反转(交换)栈顶的两个值
+	弹出两个值，相加，然后将和压入到栈中
-	弹出两个值，相减，然后将差压入到栈中
*	弹出两个值，相乘，然后将积压入到栈中
/	弹出两个值，相除，然后将商压入到栈中
%	弹出两个值，相除，然后将余数压入到栈中
~	弹出两个值，相除，将商压入到栈中，然后再将余数压入到栈中
^	弹出两个值，第一个数是第二个数的幂指数，然后将结果压入到栈中
v	弹出一个值，求平方根，然后将结果压入到栈中
k	弹出一个值，使用这个值设置精确度

图 8-3 dc: 最重要的命令

提示

如果喜欢数学或者科学思考，那么您可以发现使用 **dc** 非常有乐趣。这样做时，您将发现 **dc** 不仅仅是消遣。为了学习如何使用该程序，您需要掌握逆波兰表示法的思想，并且还要学习如何使用栈，而这两个概念都是不容易理解的概念。但是，一旦掌握了这些思想，您将发现 **dc** 是一个高效、设计良好的工具，非常易于使用。

如果再回到第 1 章的末尾，那么您将看到我对 Unix 所做的两个一般性注释：Unix 易于使用，但是难于学习。您需要首先从基本的知识开始学习，然后再根据自己的需要，学习其他的知识。

您能看出 **dc** 正好适合这种范式吗？**dc** 的使用非常简单，但是学习比较困难。您开始时学习基本知识(阅读本章内容)，然后再根据自己的需要去体验它。

因此，如果有时间练习使用 **dc**，那么您不仅在学习如何使用一个有趣的工具，而且还在训练自己以 Unix 的方式思考。

8.17 练习

1. 复习题

1. 描述 3 种用来停止程序的不同方式。
2. 哪个程序可以用来显示时间？日期呢？
3. **cal** 程序和 **calendar** 程序之间有什么区别？
4. 如何显示正在使用的计算机的名称？操作系统呢？用户标识呢？

2. 应用题

1. 米老鼠(Mickey Mouse)诞生于 1928 年 11 月 18 日。在系统中显示这个月的日历。米老鼠诞生在星期几？这一天是这一年的第几天(1 月 1 日=1, 12 月 31 日=366)？

2. 元素镱是一种银白色的非常稀有的重金属。它是世界上最贵重的金属。Gaipajama 王公希望您担任他儿子的临时保姆。报酬是每 2 小时 1 克镱，而且他希望您工作 5 小时。假定黄金的价格是每克 25.42 美元，而且 1 克镱值 6 克黄金。使用 **bc** 程序计算您做临时保姆可以挣多少美元。答案必须精确到小数点后 2 位(完成计算后，您可以在 Internet 上查找一下 Gaipajama 王公)。

3. 思考题

1. 在本章的开头，我做了一个这样的注释：“传统上讲，人们学习 Unix 的方式之一就是寓学于乐。”但是，对于其他操作系统(如 Microsoft Windows)来说，情况并不总是如此。为什么呢？

2. **users**、**who** 和 **w** 命令都可以显示当前登录到系统的用户标识的信息。我们需要使用这 3 种不同的程序吗？



文档资料：Unix 手册与 Info

在 Unix 世界中，有许多不同的文档资料系统，每一种系统都拥有自己的特点。其中一些文档资料系统被广泛使用，而另一些系统只服务于特定的领域。通常，所有这些系统都拥有两个共同的目标：使程序员易于存档他们的工作；使用户易于学习如何使用程序员创建的工具。

在本章中，将讲授如何使用两个最重要的 Unix 文档资料系统：Unix 联机手册(每个 Unix 系统都提供的一项功能)和 Info(GNU 项目的官方文档资料系统)。

这两个工具都要在 Unix CLI(命令行界面)中使用。原因在于图形化程序都是自文档化的，在某种意义上它们几乎都拥有自己的内置帮助功能。因此，当需要基于 GUI 的程序的帮助功能时，不需要使用联机手册或 Info。在程序本身中就可以找到帮助内容——通常是打开 Help 菜单。

9.1 Unix 传统与自学

正如第 2 章中所讨论的，Unix 于 20 世纪 70 年代初期在新泽西州的贝尔实验室(那时属于 AT&T 公司)开发。在 Unix 创建后不久，它就在程序员和研究人员中流行起来，首先是在贝尔实验室，然后扩展到许多大学的计算机系。

随着 Unix 的日益流行，越来越多的人需要学习如何使用该系统。但是，贝尔实验室的程序员都是大忙人，他们没有时间，也没有这种意向去教新用户如何使用 Unix。此外，Unix 的流行文化鼓励任何人都去创建新工具，并与其他用户共享。因此，随着时间的推移，以及新用户的增加，人们需要学习的内容也快速递增。

为了响应这些需求，Unix 开发人员采纳了两种解决方案。首先，他们创建了一种联机手册，并内置在 Unix 中，该手册中包含每个 Unix 工具的信息。因为 Unix 手册本身就是 Unix 的一部分，所以用户可以随时使用它。这意味着，当一名位于偏远位置的用户在深夜碰到问题时，他可以求助于该手册寻求帮助。

解决方案的第二部分就是鼓励建立一个工作环境，在这个工作环境中所有用户——不管是新用户还是有经验的用户——都需要在请求帮助之前尽自己最大的努力解决自己的问题。更准确地说，我们所谓的 Unix 传统就是要求您自己学习，尽力自己解决问题。但是，

如果您尽了最大的努力,而问题依然存在,那么其他 Unix 用户将乐意帮助您。反过来说,一旦您成为一名有经验的用户,那么就要求您帮助他人。

Unix 这一传统之所以重要的原因有两方面。首先,它提供了一种有效扩散 Unix 的方式。因为只有在真正需要时人们才请求帮助,所以有经验的 Unix 用户就不必在没有必要提供帮助的用户身上浪费时间。其次,通过使人们自学 Unix,可以鼓励 Unix 开发人员的独立思考和个人创新,从而使 Unix 更加繁荣。实际上,Unix 传统培养了一代聪明、独立并且乐于助人(当需要时)的用户,他们在一个协作创新的氛围中工作。

例如,如果一名程序员需要一个新工具,Unix 文化将鼓励他自己创建这个新工具。一旦这个程序完成,这个新程序就可以添加到普通 Unix 系统中。然后程序员编写相关的文档资料,并添加到联机手册中。在任何希望学习如何使用这个新工具的用户将阅读联机手册、体验并自学的前提下,这个新工具通过普通 Unix 社区发布出去。如果用户发现了 bug 或者严重的问题,那么该用户可以自由地与该程序的作者联系。

因此,可以基于下面两个主要思想理解 Unix 传统:在请求帮助之前尽最大的努力自己解决问题;当他人请求您的帮助时,心甘情愿地帮助他人。事实证明这些思想非常重要,最终它们被抽象为一个单独的非常古怪的单词:RTFM。

名称含义

联机/在线

以前,单词联机/在线(online)用来描述连接到一个特定的计算机系统上这一情形。例如,当您登录到系统以后,我们就可以说您联机了。

当讨论 Unix 联机手册时,我们正是以这种方式使用该单词的。手册是“联机”的,这是因为它对 Unix 系统的所有用户可用。

现在,我们还使用术语“联机/在线”表示资料或人连接到 Internet 上,而不再是一台具体的计算机系统。例如,一旦您连接到网络上,就可以使用在线银行、在线支付,甚至还可以进行在线联系。

因此,作为一名 Unix 用户,您以两种不同的方式联机:登录到一台特定的 Unix 系统上,或者连接到 Internet 上。

9.2 RTFM

单词 RTFM 由于下述几个方面而显得非常独特。首先,它是英语中没有元音的最长动词。其次,它通常全部按大写字母拼写。最后,因为 RTFM 没有元音,所以这个单词发音成 4 个不同的字母(“R-T-F-M”),虽然这个单词并不是一个只取首字母的缩写词。

正如前面所述,RTFM 是一个动词(稍后我将解释它的来源)。我们使用它来体现这样一种思想,即在请求帮助或者信息前,必须尝试着自己解决问题或者寻找信息。

单词 RTFM 有两种使用方式。首先,您可以告诉他人不要打扰您,除非他自己已经尽力了。例如,如果有人说:“您可以向我示范如何使用 **whatis** 命令吗?”您可以回答:“RTFM。”在这种情况下,RTFM 意味着“除非您查看了 Unix 联机手册,否则不要请求

帮助。”

其次，您可以使用 RTFM 表示在请求帮助之前，您已经自己试图解决这个问题了。例如，您可以向一位朋友发送一封电子邮件：“您能帮我设置 Linux 系统与 Windows PC 共享文件吗？我已经 RTFM 两天了，仍然没有办法实现这个目的，只能每隔几小时重新启动一下 Windows。”

从一开始，RTFM 思想就是 Unix 文化的一部分。现在，它的应用已经扩展到 Internet 上，特别是在 Usenet 上和开放源代码社区中(有关开放源代码运动的讨论，请参见第 2 章)。随着 RTFM 应用的日益扩大，它的含义也在不断扩大。现在，RTFM 意味着不仅要求在 Unix 联机手册中，而且还要求在 Internet 上查找信息。

因此，除非至少使用了一个搜索引擎(例如 Google)查找了相关网站，否则最好不要请求帮助。至于 Usenet，如果您是讨论组的一名新手，那么在发送第一个帖之前希望您阅读一下该组的 FAQ(frequently asked question list, 常见问题解答)。RTFM 就是这样考虑的。

提示

当查找 Unix 问题的解决方法时，不要忘记了使用 Usenet，这是一个世界范围内的讨论组系统。

搜索 Usenet 最简单的方法就是使用 Google 的 Usenet 存档，称为 Google Groups。我经常搜索 Usenet 存档，即使是最晦涩的问题，也经常能在这里找到答案。

如果在进行了所有搜索之后，还找不到希望的答案，那么您就可以在合适的讨论组中提交一个请求。如果您这样做，请一定要提及您已经 RTFM 了。

名称含义

RTFM、RTFM'd

RTFM 是一个动词，表示这样的思想，即当您需要信息或者在解决问题时，在请求他人帮助之前，应该花一些时间尝试自己去寻找所需的东西。

当谈及已经执行过这样的动作时，我们使用 RTFM 的过去分词，并拼写成 RTFM'd，而不是 RTFMed。因此，您可以说：“I have RTFM'd for the last two hours, and I can't figure out how to connect my cat to the Internet(我已经 RTFM 了两个小时，可是我还是无法知道如何将猫连接到 Internet 上)。”

和许多技术单词一样，RTFM 刚开始时是一个只取首字母的缩写词。在 Unix 的早期时代，RTFM 代表“Read the fucking manual(阅读该死的手册)”，当然，这里指的是 Unix 联机手册。但是，现在 RTFM 已经不是一个取首字母的缩写词，而是凭自己的资格成为一个合法的单词。

这并不稀奇。许多技术术语都是这样的，例如 radar(radio detection and ranging)、laser(light amplification by stimulated emission of radiation)和 scuba(self-contained underwater

* 有时候，您可能看到 RTFM 使用委婉的方式解释，即“Read the fine manual(阅读精美的手册)”。但是，您知道，在本书中，按照事情真正的本意解释是我的一贯做法。在这个例子中，您可以看出，RTFM 的原本含义含有亵渎的意味，但是我认为您应该知道真相。谢谢大家没有因此攻击我。

breathing apparatus), 还有各种专有名词, 例如 Nato(North Atlantic Treaty Organization), 以及集合名词, 例如 yuppie(young urban professional)。

RTFM 和其他这类单词之间最大的区别是 RTFM 通常都拼写成大写字母。这只能这样解释, 即在我们的文化中 RTFM 要比 radar、laser、scuba、Nato 或 yuppie 都更加重要。

9.3 什么是 Unix 手册? man

Unix 手册通常称为联机手册, 或者更简单一些, 称为手册, 它是一个文件集, 其中每个文件都包含一个具体 Unix 命令或者主题的相关文档资料。该手册在每个 Unix 系统上都提供, 所有用户都可以在任何时间使用它。要使用该手册, 只需输入一个简单的命令(我们稍后讨论)。然后您所请求的信息就会每次一屏地呈现给您。

以前(在大型计算机时代), 大多数计算机系统都提供有大量打印的高技术文档资料, 这些文档资料通常在一个中心位置(例如计算机房或者终端室)集中保存。这种文档资料不仅不方便使用, 而且还经常过期, 同时需要处理一大堆的打印更新。基于这一原因, 旧式的计算机手册要存放在一个笨拙的大夹子中, 从而在需要时可以打开插入新页, 但是它使用起来或者从一个地方搬到另一个地方都比较累赘。

Unix 就不同了。从一开始, 文档资料就始终联机, 这意味着无论何时, 所有用户在自己的终端上都可以比较方便地阅读任何方面的内容。此外, 因为联机手册存储为一组磁盘文件, 所以添加新文件就可以添加新的资料, 修改文件就可以更新已有的资料, 这都相当简单。刚开始时, Unix 系统既提供打印的手册, 也提供联机手册。打印手册中的信息与联机手册中的信息相同, 但是联机手册中的信息相对来说要提供得更早。

Unix 手册的访问相当容易。您所需的全部动作就是输入单词 **man**, 后面跟着希望了解的命令的名称。Unix 将显示这个命令的文档资料。

例如, 要显示 **cp**(复制文件)命令的文档资料, 可以输入:

```
man cp
```

假如希望了解 **man** 命令本身, 则可以输入:

```
man man
```

提示

man 命令是一个最重要的 Unix 命令, 因为使用它可以学习其他任何命令。

如果希望学习多条命令, 可以在同一行上输入希望学习的命令的名称。例如:

```
man cp mv rm
```

Unix 将逐个显示每条命令的文档资料。顺便说一下, 这 3 条命令用来复制、重命名和删除文件。第 25 章中将正式地介绍它们。

名称含义

联机手册

Unix 手册一向以来都是比较重要的。实际上，有一段时间，当 Unix 还基本上是 AT&T 公司贝尔实验室的产品时，有一批 Unix 版本是根据手册的当前版本命名的，例如 Unix Sixth Edition 和 Unix Seventh Edition 等。

尽管有关 Unix 方面的书籍和参考资料有许多，但是当人们指“手册”时，您可以假定他所指的就是那个唯一的 Unix 联机手册。例如，假设您正在阅读一封来自系统管理员的电子邮件，在电子邮件中系统管理员说他刚在系统上安装了一个新程序。在该消息的末尾，他说：“更多的信息，请查看手册。”不必问，您就可以假定，他希望您使用 `man` 命令阅读联机手册中的合适条目。

对 Unix 用户来说，毫无疑问所谓的手册就是 Unix 联机手册。

9.4 说明书页

刚开始，Unix 用户使用慢速的终端将输出打印在纸上。因为那时还没有显示器，所以当有人希望学习一条命令时，他不得不打印联机手册中的相关页。当时，这并没有像听起来那么不方便，因为那时候手册中还没有那么多的条目，而且许多条目还进行了特别设计，从而只需使用一张纸。

现在，Unix 手册中拥有大量的条目，其中许多条目已经不能在一张纸上打印。然而，无论条目有多长，习惯上还是称一个单独的条目为一页，或者更正式地称之为一个说明书页。例如，Bash(Linux 默认 shell，我们将在第 12 章中讨论)的文档资料就超过了 4500 行。然而，还是称它为一个单独的说明书页。

体会一下下面例子的含义。您到一个 Unix 吧中，喝一杯含咖啡因的巧克力牛奶，并碰巧听到两名程序员在谈话。第一名程序员说：“我不知道在情人节送我女朋友什么礼物。您有什么建议吗？”另一名程序员回答到：“为什么不给她打印一份 Bash 的说明书页呢？”

9.5 显示说明书页

实际上联机手册中的几乎每个条目都无法一屏显示。如果一次全部显示一个条目，那么大部分的内容将在您还没来得及阅读的情况下滚动出计算机屏幕。

这是一种常见的情形，不过 Unix 有一种好的解决方法：将输出发送到一个程序中，再让这个程序一次一屏地显示输出。经常在 Unix 系统中使用的这类程序有 3 个，通常称它们为分页程序。这 3 个程序分别是 `less`、`more` 和 `pg`。最好也是使用最广泛的分页程序是 `less`，我们将在第 21 章中详细讨论该程序。现在，只是作一个简要概述，从而使您能够使用它们来阅读联机手册。

如果您希望在系统上进行练习，那么您可以输入下述命令之一，每条命令都会显示一

个特定 shell(Bash、Korn shell 或者 C-Shell)的信息:

```
man bash
man ksh
man csh
```

我的建议是显示您准备使用的或者大多数人在您的系统上使用的 shell 的说明书页。如果无法确认是哪个 shell,那么您可以随便挑一个——这只是为了练习。

分页程序的任务是每次一屏地显示数据。一屏显示满了之后,程序就会暂停并且在屏幕的左下角显示一个提示。根据分页程序的不同,提示的内容也有所不同。

less 和 **pg** 程序显示一个冒号:

```
:
```

在一些系统上, **less** 显示一个消息,而不是冒号。例如:

```
byte 1357
```

在这个例子中, **less** 告诉您它已经显示的字符数量为 1357 个(每个字节可以存放一个字符)。当继续查看文件后面的页时,该数量将递增,从而可以使您粗略地知道,从开头开始,已经查看了多少内容。

more 程序显示一个包含单词“More”的提示。例如,您可能看到:

```
--More-- (10%)
```

这意味着下面还有更多的内容,您已经查看了 10%的页面内容。

一旦看到提示,您就可以通过按<Space>键(对于 **pg**,要按<Return>键)显示下一屏信息。当您结束阅读时,按下 **q**(字母“q”)键就可以退出。

在阅读说明书页时,有许多命令可供使用。但是,通常不需要它们。大多数时候,只需简单地按<Space>键,阅读下一屏内容。当到达页面的末尾,或者找到自己希望的内容时,可以按下 **q**(字母“q”)键退出。

有时候,您可能希望使用其他一些命令,因此我准备花一点时间描述一下自己发现的最有用的命令。这些命令如图 9-1 中所示。正如前面所述,命令有许多,多得超出您的需要。图 9-1 中的命令是针对使用 **less** 命令的系统的。如果系统使用的是 **more** 或者 **pg**,则可能会有些不同。如果遇到了问题,可以使用 **h** 命令请求帮助。

对于任何程序来说,最重要的命令就是显示帮助信息的命令。在这种情况下,您只需按 **h**(字母“h”)键就可以显示帮助信息。当您这样做时,说明书页信息将被所有分页命令的摘要信息取代。当结束阅读帮助信息时,按下 **q** 就可以退出,并返回到说明书页中。注意,摘要信息也相当长,和说明书页本身一样,浏览信息时必须按<Space>键才行。但是,最重要的命令都位于摘要信息的顶部。

下面准备讨论的命令是针对 **less** 分页程序的,因为大多数 Unix 系统使用它。如果 **man** 命令使用的是 **more** 或者 **pg**,那么为了获得该特定分页程序的帮助摘要信息,您也只需按下 **h** 键。我的建议就是,在阅读本书的过程中,使用计算机试验各种不同的命令。

通用命令	
q	退出
h	显示帮助信息
阅读说明书页	
<Space>	显示下一屏
<PageDown>	显示下一屏
f	显示下一屏
<PageUp>	显示上一屏
b	显示上一屏
搜索	
/pattern	向下搜索特定的模式
?pattern	向上搜索特定的模式
/	向下搜索上一模式
n	向下搜索上一模式
?	向上搜索上一模式
N	向上搜索上一模式
在说明书页中移动	
<Return>	向下移一行
<Down>	向下移一行
<Up>	向上移一行
g	移到页的顶部
G	移到页的底部

图 9-1 阅读说明书页：重要的命令

下面开始，假设您正在查找一个特定的模式，输入/(斜线)字符，再键入模式，然后按下<Return>键。例如：

```
/output<Return>
```

这个例子告诉分页程序直接跳到下一个包含单词“output”的行上。一旦指定了一个模式，就可以通过输入字符/本身再次搜索该模式。

```
/<Return>
```

如果您在搜索一种模式，但是搜索到的行并不是希望的那一行，那么您可以继续搜索相同的模式，一遍又一遍地进行，直至找到希望的那一行。这时只需不停地输入 `/<Return>`^{*}。

另外，也可以通过按下 **n**(next, 下一个)键一次或者多次搜索同一个模式。

如果要向后搜索，可以使用 `?` 来取代 `/`。例如：

```
?output<Return>
```

如果要向后搜索相同的模式，可以使用 `?` 本身：

```
?<Return>
```

另外，也可以按下 **N**(next, 下一个)键向后搜索同一个模式。

您已经知道，为了向下移动一屏，只需按 `<Space>` 键即可。您还可以按 **f**(forward, 向前)键向下移动一屏。为了向上移动一屏，可以按下 **b**(backward, 向后)键。另外，也可以通过按 `<PageDown>` 键和 `<PageUp>` 键向下或向上移动。

为了一行一行地向下移动，可以使用 `<Return>` 键或者 `<Down>` 键(也就是向下箭头键)。为了一行一行地向上移动，可以使用 `<Up>` 键(也就是向上箭头键)。

为了跳到页面的顶部，可以使用 **g**(“go to top”，跳到顶部)键。为了跳到页面的底部，可以使用 **G**(“go to bottom”，跳到底部)键。

9.6 两个有用的说明书页技术

到目前为止，我们所讨论的阅读说明书页的方式，其概念都相当简单。使用 **man** 命令显示一个特定主题的信息，然后每次一屏地查看该信息，直到找到自己需要的内容。

这是使用标准 Unix CLI(命令行界面)阅读说明书页的常见方法。但是，如果将 **man** 命令与 Unix 工作环境集成在一起，则可以以一种更完善的方式访问说明书页。

正如第 6 章中讨论的，可以采用两种方式使用 CLI：通过终端窗口或者通过虚拟控制台。这里的建议是学习如何同时使用两个终端窗口，一个终端窗口用于任务处理，另一个终端窗口用于显示说明书页。例如，假设您正在一个终端窗口中工作，使用 **vi** 文本编辑器编辑一个文件(参见第 22 章)。这时您需要一些帮助，因此您决定查看 **vi** 的说明书页。如果在第二个终端窗口中这样做，那么您就可以同时查看说明书页和原始的窗口，如图 9-2 所示。

为了成为一名经验丰富的 Unix 用户，您需要掌握同时使用多个窗口的技能。更准确地说，您必须能够回答下面的问题：什么时候应该使用一个单独的窗口？什么时候应该使用两个窗口？什么时候使用的窗口应该多于两个？什么时候应该忘记窗口而使用虚拟控制台呢？

^{*} 这一特征取自 **vi** 编辑器，我们将在第 22 章中讨论 **vi** 编辑器。

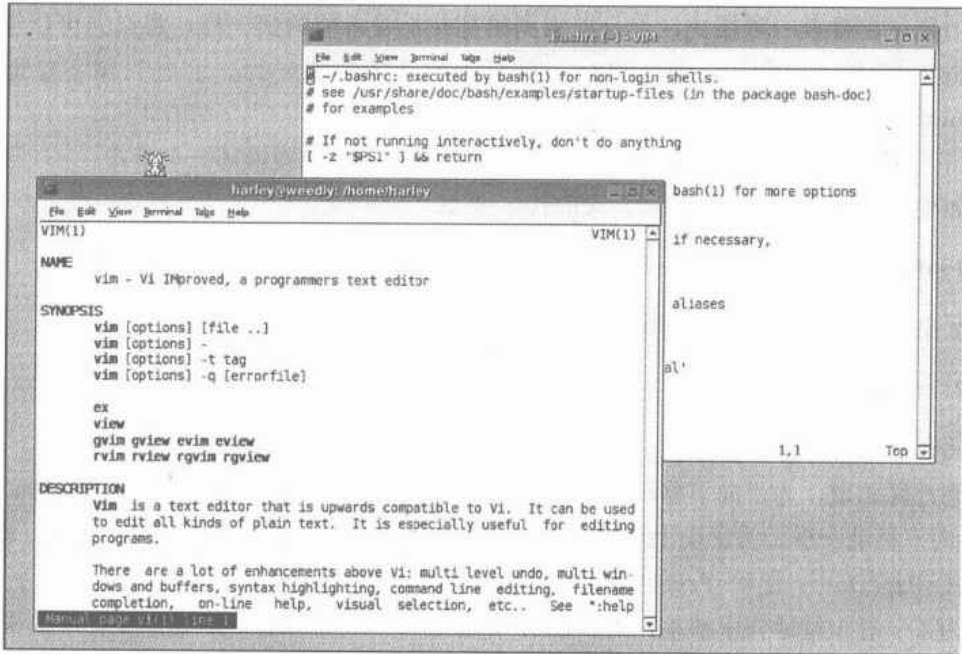


图 9-2 在单独的终端窗口中显示说明书页

通过在单独的终端窗口中显示说明书页，当您在另一个窗口中工作时，可以将说明书页窗口作为参考。在这个例子中，左边窗口是 **vim** 的说明书页，**vim** 是某种版本的 **vi** 编辑器。右边是正在运行 **vim** 的一个终端窗口。

这些问题的答案并不明显，而且您将发现随着经验的日益丰富，您的技能也会逐步增长。技巧就是永远不要墨守成规。例如，不要总是使用一个窗口，以及不要总是以完全相同的方式使用两个窗口。

为了进一步扩展您的技巧水平，我希望再向您传授一个阅读说明书页时可用的工具。

当阅读说明书页时，如果您键入一个 **!** (感叹号)，就可以在它之后键入一条 **shell** 命令。**man** 程序将把这条命令发送给 **shell**，而 **shell** 将运行这个命令。当命令结束后，可以按下 **<Return>** 键返回到 **man** 程序中。

为了明白这种方法的工作方式，我们先显示一个说明书页，然后输入：

```
!date<Return>
```

在这个例子中，我们从 **man** 程序中输入了命令 **date**。结果是显示时间和日期。一旦 **date** 命令结束，只需按下 **<Return>** 键，就会返回到 **man** 程序中的原来位置。

您可能会想，在自己希望时就能够输入一个 **shell** 命令将非常有用。特别方便的是，可以使用这一技巧在阅读一个说明书页时显示另一个说明书页，而且不必切换到一个单独的窗口。下面是它的工作方式。

假设您正在阅读 **man** 命令的说明书页：

```
man man
```

在阅读该说明书页时,您看到了几个相关的命令,其中有一个 **whatis** 命令(我们将在本章后面讨论)。此时,您决定停止正在做的事情,去阅读一下 **whatis** 的说明书页。为此,只需键入:

```
!man whatis<Return>
```

当完成 **whatis** 的阅读后,按 **q** 键退出。然后系统将要求您按<Return>键,一旦这样做了,您将返回到原始的 **man** 程序,继续阅读 **man** 命令。

名称含义

Bang

作为一名 Unix 用户,有时候可能要以一种特殊的方式使用!(感叹号)字符。它通常会修改正在进行的模式,因此可以暂停当前的程序,向 shell 发送一条命令(有关模式的思想,请参见第 6 章中的讨论)。

作为一个示例,假设您正在阅读一个说明书页,您可以输入下述命令显示时间和日期:

```
!date<Return>
```

当以这种方式使用!字符时(作为一条命令而不是一个标点符号),它拥有一个特殊的名称。我们称它为 Bang 字符,或者更简单一些,称它为 Bang。因此,Unix 人士可能会说:“如果您正在阅读说明书页,而且希望显示时间和日期,那么只需输入‘bang-d-a-t-e’并按<Return>键即可。”

名称“bang”是一个俚语,已经在印刷和排版行业应用了很长时间。它的来源不明。

9.7 man 的备选方案: xman 和 Web

正如前面所述,可以使用 **man** 命令显示 Unix 联机手册中的说明书页。除该命令之外,我还希望您知道另外两种方法。

首先,大多数 Unix 系统的说明书页都可以在 Internet 上找到。这意味着,当需要时,可以使用浏览器查找并显示一个特定的说明书页。在 Web 上阅读说明书页的优点在于它们通常是相互链接的,从而允许从一个页面跳到另一个页面。常规的说明书页(使用 **man** 命令显示的说明书页)是部分内容高亮显示的纯文本,而不是带有链接的超文本。

在 Web 上查找说明书页的最简单方法就是使用搜索引擎(例如 Google)搜索“man”加上命令的名称,例如:

```
"man whatis"
```

一定要确保包含双引号。

另外,还可以通过搜索“man pages”加上 Unix 的类型名称查找更普遍的资源,例如:

```
"man pages" Linux
```

```
"man pages" FreeBSD
```

"man pages" Solaris

再次强调，不要忘了双引号。

提示

当有时间时，查找一些为 Unix 或者 Linux 提供说明书页的网站。挑选一个自己特别喜欢的，然后保存该站点的 URL(Web 地址)，这样无论何时，当需要时都可以快速地访问该站点。

我的建议是将 URL 保存为浏览器链接栏中的一个按钮。这样的话，该 URL 就总是可见，并且易于使用。不过，您也可以将 URL 保存为桌面上的一个图标，或者保存到 Bookmarks/Favorites 列表中。选择哪一种方式关键在于它是否最适合您。

另一种使用基于 Web 的说明书页的方法是 **xman**，**xman** 是一个基于 GUI 的程序，它充当说明书页浏览器("x"暗含着该程序是为基于 GUI 的 X-Window 编写的，参见第 5 章)。如果系统上提供了 **xman**，则值得花些时间来体验以及学习如何使用它。

启动 **xman** 时，需要在命令行上输入下述命令：

```
xman &
```

使用**&**(和号)字符是告诉 shell 将程序启动为后台运行。

当 **xman** 启动后，它显示一个简单的窗口，窗口中有 3 个大的按钮，这 3 个按钮的标签分别为 Help、Quit 和 Manual Page(参见图 9-3)。相对于让我来解释该程序的全部复杂性，由您自己来阅读使用说明更简单些，因此在此只提供两个提示。

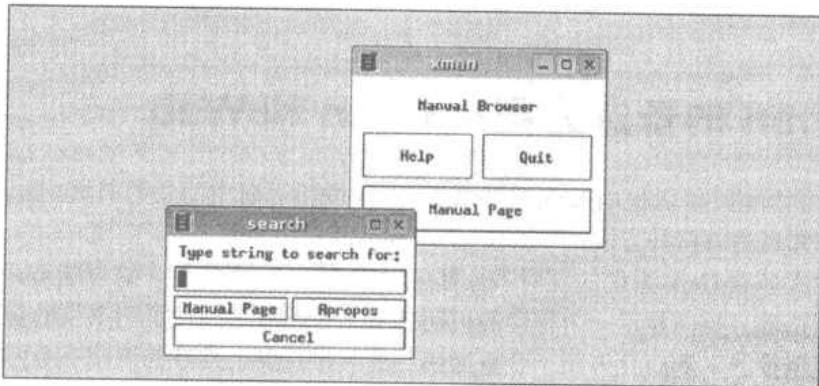


图 9-3 xman 程序

xman 是一个基于 GUI 的用于浏览说明书页的程序。在右上角是初始窗口。为了开始，需要单击“Help”。在左下角显示了一个搜索框，通过按<Ctrl-S>可以在任意时间显示该搜索框。

首先，为了开始，单击“Help”按钮，并阅读使用说明。当焦点位于 **xman** 窗口中时，按下^S(<Ctrl-S>)显示一个小的搜索框。这样就可以快速方便地查找自己希望的内容(参见图 9-3)。试试看，您就可以明白我的意思。

9.8 Unix 手册的组织方式

最好把联机手册想象成一个存在于 Unix 系统中的庞大参考书。这本书就像一本百科全书, 其中包含许多条目, 各个条目按照字母顺序排列, 并且每个条目都涵盖一个单独的主题。

这本书无法翻页, 因此, 该书没有页号, 也没有正式的目录表或者索引。但是, 该书采用适合于电子书的组织形式, 分成了若干个层次。

传统上, 整个手册分成 8 节, 编号从 1 至 8。这些经典的分类如图 9-4 所示。各个系统之间, 实际名称可能有所不同, 但是, 在很大程度上, 所有的 Unix 手册都倾向于遵循相同的组织原则。尽管您的系统上的手册可能有点不同, 但是它应该很接近图 9-4, 因此, 当阅读下述讨论时, 您应该能够理解。

1. 命令
2. 系统调用
3. 库函数
4. 特殊文件
5. 文件格式
6. 游戏
7. 杂项信息
8. 系统管理

图 9-4 Unix 联机手册的 8 节内容

手册中最重要的部分就是第 1 节。该部分包含大量 Unix 命令的说明书页。实际上, 除非您是一名程序员或者系统管理员, 否则您或许只需了解手册的这一部分内容。

如果您是一名程序员, 那么您也会对第 2 节和第 3 节感兴趣。第 2 节包含系统调用的说明书页, 系统调用在程序中使用, 用来请求内核执行一个特定的动作。第 3 节包含库函数的文档, 库函数有时候称为子例程。这些是标准化的工具, 不直接涉及内核, 用来在程序中执行特定的任务。

第 4 节讨论特殊文件, 即一种通常表示物理设备的文件。这一部分还包含设备驱动程序(充当设备接口的程序)的信息。本部分内容主要由程序员使用。

第 5 节描述系统使用的重要文件格式, 包括配置文件。这一部分主要由程序员和系统管理员使用。

第 6 节提供系统中所安装游戏的说明书页。以前, 主要是基于文本的游戏和娱乐活动, 用户从命令行上使用。一个典型的例子就是第 7 章中讨论的 *Rogue* 游戏。那个时代, 第 6 节是手册的重要部分。现在, 大多数系统已经删除了基于文本的游戏, 因此大多数情况下, 手册的这一部分是空的。可以肯定的是, 系统中现在有许多基于 GUI 的游戏, 但是正如本章开头所述, 这样的程序都提供了内置的帮助, 因此它们不再需要说明书页。

这并不是说已经不再有基于文本的 Unix 游戏了。基于文本的 Unix 游戏还有许多, 而

且还有许多奇妙的娱乐活动，如果您想安装的话，就可以在自己的系统上安装。如果使用的是共享系统，而且系统管理员没有安装游戏(或者系统管理员将游戏删除了)的话，那么手册的第 6 节就可能是空的。这是因为大多数系统管理员不希望用户由于能够阅读游戏的说明却不能玩游戏(就像站在毗斯迦山上的摩西眺望上帝赐给亚伯拉罕的迦南地方一样)而抱怨他。

第 7 节是杂项信息，包含各种混杂信息。第 7 节的内容根据系统的不同差别很大，因此除了下面一点——像其他大多数部分一样，本部分主要针对程序员和系统管理员之外，我无法讲太多的内容。

最后，第 8 节包含系统管理员用来执行工作所使用的所有特殊命令的说明书页。换句话说，这一部分中的命令是只能由超级用户使用的命令(有关系统管理和超级用户的讨论，请参见第 4 章)。

如果使用的是共享系统，那么您可能不用关心手册的第 8 节，这是因为有其他人进行系统管理工作。但是，如果在自己的计算机上运行 Unix，那么您就是系统管理员，有时候您需要使用本部分的一些命令。

提示

除第 1 节(命令)和第 6 节(游戏)之外，Unix 手册中的大部分内容，只有程序员和系统管理员才感兴趣。

将 Unix 手册组织成这 8 个特定节的思想起源于最早的 Unix 实现，而且多年过去之后，它在很大程度上还保持着原样。但是，现代的手册要比它们值得尊敬的祖先包含更多的资料。因此，在您的系统上，您可能看到不同的、更加复杂的部分，或许还拥有不同的名称。

每一节又被划分为几个小节。例如，在一些 Linux 系统上，在第 3 节(库函数)中，就有以下几个小节：第 3c 节是标准的 C 函数；第 3f 节是 Fortran 函数；第 3m 节是数学函数；第 3s 节是标准 I/O 函数；而第 3x 节则是特殊函数。

9.9 在 man 命令中指定节号

到目前为止，我们已经知道了如何通过键入 **man**，后面跟着命令的名称来使用 Unix 手册。例如，为了学习 **kill** 命令(参见第 26 章，该命令用来停止失控的程序)，可以输入：

```
man kill
```

这个命令将显示手册第 1 节中 **kill** 的说明书页。

但是，碰巧的是手册的第 2 节(系统调用)中也有一个 **kill** 条目。如果真正需要的是这个，可以在命令名的前面指定节号：

```
man 2 kill
```

该命令告诉 Unix 您只对手册一个特定节中的内容感兴趣。

如果使用的是一种派生于 System V(参见第2章)的 Unix 系统,那么命令的形式就有点不同:必须在节号前键入 **-s**。例如, Solaris 就是这种情况:

```
man -s 2 kill
```

如果节又划分为小节,那么也可以明确指定自己希望的小节号。

例如,在一些系统上,第 3f 节(手册中记录 Fortran 子例程的那一部分)中有一个 **kill** 命令的条目。为了显示该说明书页,可以输入:

```
man 3f kill
```

正如前面所述,您可以同时查看手册的多个部分。例如,如果希望查看 **kill** 的全部 3 个条目,可以输入:

```
man 1 kill 2 kill 3f kill
```

当没有指定节号时,Unix 从手册的开头(第 1 节)开始,一直向后寻找,直至找到第一个匹配。因此,下面两条命令拥有相同的结果:

```
man kill
man 1 kill
```

提示

大多数时候,您只对手册的第 1 节(命令)感兴趣,因此指定节号并不必要。当寻找与编程(第 2、3、4、5 和 7 节)或者系统管理(第 4、7、8 节)相关的信息时,才需要使用节号。

为了使您熟悉手册的各种不同部分,每个节和小节都包含一个称为 **intro** 的页面,该页面充当一个简介。一种熟悉本节内容的方法就是阅读它的 **intro** 页。

下面举一些显示这类页面的命令示例:

```
man intro
man 1 intro
man 1c intro
man 6 intro
```

众所周知,在默认情况下 **man** 假定您希望参考第 1 节,因此前两个例子是等价的。

提示

如果您是一名初学者,那么最好使用下述两个命令来学习联机手册:

```
man intro
man man
```

9.10 说明书页的引用方式

在阅读 Unix 手册的过程中,经常会看到命令名的后面跟有一个用括号括起来的数字。这个数字用来说明在手册的哪一节中查找该命令的信息。

例如,下面是一段从 BSD(伯克利)版本的 **chmod** 命令(将在第 25 章中讨论)的说明书页中摘录的句子。现在,不用关心该句子的含义,只是看看引用的命令:

“...but the setting of the file creation mask, see **umask(2)**, is taken into account...”

“**umask(2)**”说明可以在手册的第 2 节中查找 **umask** 的说明书页。如果要阅读这部分的信息,可以使用:

```
man 2 umask
```

因为已经知道第 2 节内容描述的是系统调用,所以我们可以猜到只有在编程时才会用到这一部分。

但是,在 **chmod** 命令说明书页的最后,有这样两行:

SEE ALSO

```
ls(1), chmod(2), stat(2), umask(2), chown(8)
```

这是与 **chmod** 命令相关的其他 5 个命令的引用。可以看出,其中有 3 个引用在第 2 节中,它们是针对程序员的。最后一个引用位于第 8 节中,是针对系统管理员的。

第一个引用指向了一个命令,即 **ls**,它的说明书页位于第 1 节中。因为第 1 节描述一般命令,所以这一引用对很多人来说都极有价值。为了显示该说明书页,可以使用下述两条命令中的任意一条:

```
man ls
```

```
man 1 ls
```

顺便说一下,**ls** 命令的目的是显示文件的名称。我们将在第 24 章中讨论该命令。

提示

在查找信息或者处理问题时,如果看到了手册第 1 节中命令的引用,就应该花点时间看看该引用。即便所显示的信息并不是现在所需的信息,它至少也会对日后有利。

但是,如果看到其他节的引用,则可以忽略它们,除非该信息看起来特别吸引人。

9.11 说明书页的格式

每一个说明书页都解释一个主题,通常是一个命令、系统调用或者库函数。一些页较短,但有些页则相当长。例如,描述各种 **shell** 的说明书页都非常长,足以成为一个独立的参考手册。为了明白我的意思,可以试试下面的几条命令:

```
man bash
```

```
man ksh
```

```
man csh
```

为了方便起见,每个说明书页,无论大小都按照标准格式组织,在这种组织方式中,每个页都分成几个部分,每部分都拥有自己的标头。其中最常见的标头如图 9-5 所示。更

有趣的是, 这些标头与许多年以前, 由贝尔实验室开发的原始 Unix 手册中的标头一样(当然, 从那时起, 内容已经进行了大幅度的修改)。

标头	含义
Name	命令的名称和用途
Synopsis	命令语法
Description	完整描述(可能很长)
Environment	命令使用的环境变量
Author	程序员的名字
Files	对该命令重要的文件列表
See also	查看相关信息的位置
Diagnostics	可能的错误和警告
Bugs	错误、缺点、警告

图 9-5 说明书页中使用的标准标头

并不是所有的说明书页都包含每一个标头, 而有些说明书页所含的标头也并不位于该列表中。例如, 我就曾经遇到过 **Examples**、**Reporting Bugs**、**Copyright**、**History** 和 **Standards** 标头。但是, 不考虑实际设计, 各个说明书页的基本格式都是相同的。实际上, 我曾看到过的每个说明书页都以相同的 3 个标头 **Name**、**Synopsis** 和 **Description** 开始。

图 9-6 示范了一个说明书页样本。该说明书页实际上取自一个较早的 Unix 系统, 您的系统上等效的说明书页极有可能要比这长许多。之所以仍然使用这个特殊的例子, 是因为它足够短可以打印、易于理解, 而且包含典型说明书页中所有重要的元素。

注意, 在阅读图 9-6 中的说明书页时, 要记住单词 “print” 通常指的是在终端上显示文本, 而不是实际打印(参见第 7 章)。

在讨论之前, 首先快速地浏览一遍每个基本的标头。正如前面所述, 您有时候会看到其他标头名称, 但是一旦拥有一些经验之后, 就可以毫无困难地理解各种变体。

(1) **NAME**: 这是一个单行的命令或功能摘要。要意识到有些摘要会比较含糊, 因此如果觉得迷惑, 可以多进行一些 RTFM。

(2) **SYNOPSIS**: 本节说明命令的语法。这是如何输入命令的权威性解释。第 10 章将详细描述命令语法。因此, 我们将把大多数讨论留在第 10 章中进行。现在, 我只希望引起您的一点注意。

一般而言, 在输入命令时, 首先键入命令的名称, 后面跟着选项, 选项后是参数。我们将在第 10 章中讨论技术细节, 因此现在不用担心。我希望您理解的全部就是 **Synopsis** 节以两种变体形式说明命令选项。

首先, 您可能只看到单词 **OPTION**。在这种情况下, 实际选项在下面的 **Description** 节中列举并解释。下面举一个例子, 取自 **ls** 命令的 Linux 版本的说明书页。

ls [OPTION]... [FILE]...

GNU 实用工具(参见第 2 章)提供的说明书页使用这种约定。因为几乎所有的 Linux 系统都使用 GNU 实用工具, 所以大多数 Linux 的说明书页都是这种形式。

MAN(1)	USER COMMANDS	MAN(1)
NAME		
man - display reference manual pages; find reference pages by keyword		
SYNOPSIS		
man [-] [section] title ...		
man -k keyword ...		
man -f filename ...		
DESCRIPTION		
<p>Man is a program which gives information from the programmer's manual. It can be asked for one-line descriptions of commands specified by name, or for all commands whose description contains any of a set of keywords. It can also provide on-line access to the sections of the printed manual.</p> <p>When given the option -k and a set of keywords, man prints out a one-line synopsis of each manual section whose listing in the table of contents contains one of those keywords.</p> <p>When given the option -f and a list of names, man attempts to locate manual sections related to those files, printing out the table of contents lines for those sections.</p> <p>When neither -k or -f is specified, man formats a specified set of manual pages. If a section specifier is given man looks in that section of the manual for the given titles. Section is either an Arabic section number (3 for instance), or one of the words "new", "local", "old" or "public". A section number may be followed by a single letter classifier (for instance, lg, indicating a graphics program in section 1). If section is omitted, man searches all sections of the manual, giving preference to commands over subroutines in system libraries, and printing the first section it finds, if any.</p> <p>If the standard output is a teletype, or if the flag - is given, man pipes its output through more(1) with the option -s to crush out useless blank lines and to stop after each page on the screen. Hit a space to continue, a control-D to scroll 11 more lines when the output stops.</p>		
FILES		
/usr/man	standard manual area	
/usr/man/man?/*	directories containing source for manuals	
/usr/man/cat?/*	directories containing preformatted pages	
/usr/man/whatis	keyword database	
SEE ALSO		
apropos(1), more(1), whatis(1), whereis(1), catman(8)		
BUGS		
<p>The manual is supposed to be reproducible either on a photo-typesetter or on an ASCII terminal. However, on a terminal some information (indicated by font changes, for instance) is necessarily lost.</p>		

图 9-6 取自 Unix 手册中的样本说明书页

下面再举另外两个例子。其中第一个例子取自 FreeBSD 手册, 第二个例子取自 Solaris 手册。

```
ls [-ABCFGHLPRTWabdcdfghiklmnopqrstuwxl] [file...]
ls [-aAbcCdeEfFghHilLmnopqrRstuxl@] [file...]
```

在这个例子中, 指定了实际选项(图 9-6 中示范的样本说明书页也是这种情形)。至于之前的例子, 详细情况在 **Description** 一节中解释。**Synopsis** 的任务是提供命令的一个简单概要。

(3) **DESCRIPTION**: 该节是最大的一部分, 通常占据说明书页的主体位置。该节的目的是解释您所需知道的大多数细节, 包括如何使用选项。在一些系统上, 将完整的解释分成两个独立节: **Description** 和 **Options**。

在阅读过程中, 记住自己正在看的是一本参考手册, 而不是一本技术指南将会有所帮助。要做好有一些描述非常难懂的思想准备, 也许直到您理解了自己正在做什么才能理解这些描述。这很平常。即使遇到了困难, 也要耐心读下去, 虽说可能会有一些内容让您感到困惑。学的越多懂的也就越多, 当学习了更多知识后, 可以再去试试读这些内容。

还要意识到, 有一些描述(例如有关各种 shell 的描述)或许永远也不会完全理解。如果这使您受到了挫折, 那么要提醒自己, 那些对 Unix 手册什么都懂的人可能远不如您有魅力, 没有您那么适应社会。

(4) **FILES**: 本节说明该命令所使用文件的名称。如果本节中的信息对您没有价值, 那么可以忽略它(我们将在第 23 章中详细讨论文件的名称)。

(5) **SEE ALSO**: 这一节非常重要。它指出手册中可以查阅更多相关信息的其他地方。特别是您将看到与正在讨论的命令以某种方式相关的命令。这些引用是一种很好的学习途径。要特别注意对第 1 节中说明书页的引用。

(6) **ENVIRONMENT**: 在解释这一节之前, 先特别介绍一下有关变量的思想。

变量是一个拥有名称和值的实体。在 Unix 中, 有这样一些变量, 它们的值可被所有的程序和 shell 脚本使用(shell 脚本是一个文件, 其中包含可自动执行的一串命令)。根据上下文的不同, 这些变量还有其他几个不同的名称: 环境变量、全局变量或者 shell 变量(参见第 12 章)。根据约定, 环境变量和全局变量的名称完全由大写字母构成。

说明书页的这一节描述程序使用的环境变量。例如, **date** 命令的说明书页引用了一个称为 **TZ** 的环境变量, 该环境变量显示所使用的时区。

(7) **AUTHOR**: 开发该程序的人的姓名。GNU 实用工具的说明书页通常提供这一节的内容。这是因为运营 GNU 项目的自由软件基金会(参见第 2 章)喜欢赞扬程序员。

(8) **DIAGNOSTICS**: 本节包含两种类型的信息。第一类信息对可能的错误消息进行解释; 第二类信息是命令结束时返回的错误代码。

错误代码对程序员很重要, 特别是在程序员希望在程序中或者 shell 脚本中调用命令, 然后测试命令是否成功完成时。如果命令成功执行, 那么错误代码的值为 0(零)。否则, 错误代码值为非 0。

(9) **BUGS**: 所有的程序都有两种类型的 bug: 一类是已知的 bug, 一类是未知的 bug。最初的 Unix 开发人员已经意识到没有程序是完美的, 用户应该知道程序的不足之处。因此, 许多说明书页中专门有一节致力于记录已知的问题。

一些商业 Unix 厂商认为将这一节命名为 **Bugs** 可能会让付费客户产生误解。因此, 您

可能会发现这一节被修改为一个不太明显的名称, 例如 **Notes** 或者 **Limitations**。别被忽悠了, bug 就是 bug, 如果使用这个程序, 那么您有权知道这个程序的 bug。

9.12 一种快速查寻命令作用的方法: **whatis**

当输入 **man** 命令时, Unix 将显示整个手册页。但有时候, 您可能只对一个简要描述感兴趣。在这种情况下, 还有另外一种方法。

正如前面所述, 说明书页的 **Name** 节中包含有一行描述。如果只想看这一行内容, 可以键入 **man -f**, 后面跟一个或者多个命令的名称。例如:

```
man -f time date
```

在 **man** 命令的这种形式中, **-f** 称为一个选项(我们将在第 10 章中讨论选项)。字母 **f** 代表单词 “files”。每个说明书页都存储在一个单独的文件中。当使用 **-f** 选项时, 就是告诉 **man** 查找哪些文件。

为了方便起见, 可以使用命令 **whatis** 来取代 **man -f**。例如, 如果想显示时间, 但是不能确定是使用 **time** 命令还是 **date** 命令, 就可以输入下面两条命令中的任意一条:

```
whatis time date
man -f time date
```

您将看到类似于下面的显示信息:

```
date (1) - print or set the system date and time
time (1) - run programs & summarize system resource usage
time (3) - get date and time
time (7) - time a command
```

最后两行不是指向第 1 节的, 所以可以忽略这两行。查看头两行, 就可以知道您需要的命令是 **date**。实际上, **time** 命令是用来测量一个程序或者一条命令执行了多长时间的。

众所周知, 在输入 **man** 命令时, 可以指定一个特定的节号(例如 **man 1 date**)。至于 **man -f** 或者 **whatis** 命令, 就不能再指定具体的节号。Unix 总是搜索整个手册。

因此, 查找手册包含什么内容最好输入:

```
· whatis intro
```

这样将会显示每个 **intro** 页的简要说明(试一试)。

注意, 要想 **whatis** 命令正常运行, 说明书页必须以某种特定的方式进行预处理。这包括收集所有的单行描述, 并将它们存储在特定文件中。**whatis** 命令搜索的正是这些文件, 而不是实际手册(这种方式太慢了)。如果没有执行预处理, 那么 **whatis** 命令就不会返回有用的信息。如果您的系统出现了这种情况, 应该向系统管理员报告。

9.13 搜索命令: apropos

当希望学习某条具体的命令时, 可以使用 **man** 来显示该命令的说明书页。但是, 如果您知道想做什么, 但是却不能确定使用哪条命令, 该怎么办呢?

解决方法就是使用带有 **-k** 选项的 **man** 命令。这样将搜索 NAME 节中包含特定关键字的命令(字母 **k** 代表 “keyword”)。例如, 假如您希望查找手册中与手册自身相关的所有条目, 则可以输入:

```
man -k manual
```

为了方便起见, 可以使用单个单词 **apropos** 代替 **man -k**:

```
apropos manual
```

注意, 当发音 **apropos** 时, 重音位于最后一个音节上, 而且 “s” 不发音, 所以 **apropos** 的发音为: a-pro-poe。这是因为这一名称来自法语, 而在法语中, 单词后面的 “s” 通常不发音*。

apropos 命令搜索所有的单行命令描述, 查找那些包含有指定字符串的描述。为了使该命令功能更强大, 在该命令中 Unix 不区分大小写字母:

下面是上例的一些示例输出:

```
catman (8) - create the cat files for the manual
man (1) - displays manual pages online
man (5) - macros to format entries in reference manual
man (7) - macros to typeset manual
route (8c) - manually manipulate the routing tables
whereis (1) - locate source, binary, or manual for program
```

注意到其中有两条我们感兴趣的命令, 它们是 **man** 和 **whereis**, 也是仅有的位于第 1 节中的两条命令。同样注意到 **route** 命令也被引用, 这是因为该命令的描述中巧恰有字符串 “manual” 出现。

您或许会问, 为什么 **apropos** 和 **whatis** 没有出现在该列表中呢? 无论如何, 它们毕竟是帮助访问联机手册的命令。为了回答这个问题, 请输入:

```
whatis apropos whatis
```

您将发现单词 “manual” 并没有出现在这两条命令的描述中:

```
apropos (1) - locate commands by keyword lookup
whatis (1) - display command description
```

这里的经验是: **apropos** 命令并不神秘, 它只是盲目地查找匹配的字符串, 所以如果没有找到需要的东西, 可以换另一种方式再试试。

* 法国人擅于拼写, 但是不擅于发音。

提示

大多数命令实际上就是程序。例如，**man** 命令实际上就是一个名为“man”的程序。但是，有些最基本的命令是由 shell 本身执行的，这些命令称为内置命令。这些命令记录在 shell 的说明书页中。手册中没有它们各自的单独条目。

如果查找一条您知道存在的命令，但是根据其名称并没有找到该命令，那么可以查查 shell 的说明书页：

```
man bash
man ksh
man csh
```

如果您是 Bash 用户，还可以使用一个列出所有内置命令的特殊说明书页：

```
man builtin
```

名称含义

apropos

在 Unix 中，**apropos** 命令可以取代 **man -k**。该单词来源于法语 à propos，含义为“related to(与……相关)”。在英语中，“apropos”是一个前置词，含义为“concerning(关联)”或者“with reference to(关于)”。例如，您可能在一篇小说中读到下面一段文字：

“...Amber raised her eyebrows and reached over to touch the tall, handsome programmer lightly on the lips. As she shook her long blond hair, she felt a frisson of desire travel through her lean, lissome body. ‘Apropos to your proposal,’ she cooed seductively, batting her eyelashes, ‘I’d love to be the Mistress of Ceremonies at your Unix bachelor party. But does Christine know about the invitation?’...(Amber 扬起她的眉毛，踮起脚来，轻轻地触摸高大英俊的程序员的嘴唇。当她晃动她那金黄色长发时，感觉到一阵激流穿越她那瘦弱柔软的身体。‘关于您的建议’，她眼睛一眨一眨，充满诱惑地嚶嚶低语，‘我希望成为您的 Unix 单身汉派对上的典礼女主人。但是 Christine 知道这个邀请吗？’)”

9.14 foo、bar 和 foobar

有两个非凡的单词 **foo** 和 **bar**，您可能会时不时地看到。这两个单词由程序员使用，用作通用标识符(generic identifier)。不仅在 Unix 和 Linux 世界中，而且在 Web 和 Usenet 讨论组中也可以看到它们。

当您希望引用一个没有名称的东西时，就可以称它为“foo”；当您希望引用两个没有名称的东西时，可以称它们为“foo”和“bar”。没有人知道这一传统是从什么时候开始的，但是人们都大量使用它。

例如，下面一段文字摘录自 **exec** 命令的 Linux 说明书页(不用关心这句话的意思)。

“...Most historical implementations were not conformant in that **foo=bar exec cmd** did not pass **foo** to **cmd**...(大多数没有遵循 **foo=bar exec cmd** 形式的历史实现没有将 **foo** 传递给 **cmd**)”。

有时候,可能还会看到以这种方式使用 **foobar**。例如,下面是一位有名的 Unix 教授在一次期末考试中写的一个问题(再次强调,这里不用关心这句话的意思)。

“...Give a Unix command, and the equivalents in **sed** and **awk** to achieve the following: Print the file named **foobar** to standard output, but only lines numbered 4 through 20 inclusive. Print all the lines in file **foobar**, but only columns numbered 10 through 25 inclusive...(给出一条 Unix 命令,以及 **sed** 和 **awk** 中的等价命令实现下述操作:将命名为 **foobar** 的文件显示到标准输出中,但是只包含第 4 行和第 20 行。显示 **foobar** 文件中的所有行,但是只包含第 10 列和第 25 列。)”

名称含义

foo、**bar**、**foobar**

在 Unix 世界中以及 Internet 上,单词 **foo**、**bar** 和 **foobar** 通常用作通用术语,表示讨论或者讲解中没有命名的项。那么这些奇怪的单词来自哪里呢?

单词“**foobar**”派生于只取首字母的缩写词 **FUBAR**,该词在第二次世界大战中广为流行。**FUBAR** 的含义是“fouled up beyond all recognition(搞得一团糟而无法识别)”^{*}。

单词“**foo**”看上去有一个更加确切的历史。毫无疑问,**foo** 的流行多数起源于 **foobar**。然而,**foo** 单词的使用看上去可能更早。例如,在一部 1938 年的卡通片中,**Daffy Duck** 举着一个上面写着“Silence is Foo”的标牌(这完全正确)。一些专家推测 **foo** 可能来源于依地语中的“feh”和英语中的“phoo”。

9.15 Info 系统

Info 系统是一个联机帮助系统,独立于 Unix 手册,用来记录 GNU 实用工具(在第 2 章中解释过)。因为许多类型的 Unix——包括几乎所有的 Linux 系统,都使用 GNU 实用工具,所以大多数人发现了解如何同时使用联机手册和 **Info** 非常有用。实际上,您将会发现许多 Linux 的说明书页指向了 **Info**。

从表面上看,**Info** 与联机手册有点相似。信息存储在文件中,每个文件一个主题,这与说明书页相似。这些文件称为 **Info 文件**,而且要阅读它们时,需要使用 **info** 程序。为此,只需键入 **info**,后面跟着命令的名称即可。

考虑下面两个例子。其中第一个例子显示 **date** 命令的说明书页。第二个例子显示同一个命令的 **Info** 文件。

```
man date
info date
```

^{*} 实际上, **fubar** 中的 **F** 并不代表“fouled”。但是,我认为在同一章中两次使用“fuck”这个单词(参见 **RTFM** 的讨论)显得有点过于无礼。

我确信您可以理解。

类似于联机手册，**info** 命令也将每次一屏地显示信息。同时，按<Space>键向前移动一屏，按 **q** 键退出。但是，稍后将会看到，相似之处就这些，再也没有了。

如果在启动 **Info** 时遇到了困难，可以通过查找 **info** 程序检查您的系统上是否有该程序。下面任何一条命令都可以完成这个任务(参见第 7 章)：

```
which info
type info
whence info
```

另外，您还可以查看 **info** 的说明书页：

```
man info
```

如果您的系统上没有 **info** 程序或者没有 **info** 的说明书页，那么可以假定您的系统上没有 **Info**(这意味着您可以跳过本章的剩余部分，而对您的生活没有什么影响)。

众所周知，所有的 Unix 和 Linux 命令都拥有说明书页。但是，有许多命令没有 **Info** 文件。基于这一原因，如果试图显示一个没有 **Info** 文件的命令的 **Info** 文件，那么 **Info** 会显示该命令的说明书页。例如，**man** 命令就没有 **Info** 文件。您可以看看输入下述命令会发生什么情况：

```
info man
```

Info 和联机手册之间有 3 个主要的区别。首先，**Info** 文件不仅包含信息，而且还包含连接到其他 **Info** 文件的链接。从某种意义上讲，阅读 **Info** 页与阅读 Web 页面相似，因为您可以使用链接从一个 **Info** 文件跳到另一个 **Info** 文件*。而说明书页不是这样的。

其次，在查看 **Info** 文件时，有许多命令可以使用，要比联机手册中可用的命令多得多。这样就生成一个功能非常强大的环境。基于这一原因，有些人宁愿使用 **info** 代替 **man** 来查看说明书页。

最后，正如本章前面所描述的，联机手册是由贝尔实验室那些最初的 Unix 开发人员设计的。他们的目标就是使事情对于创建文档资料的程序员以及阅读这些文档资料的用户来说尽量简单。而 **Info** 是由 Emacs 文本编辑器的开发人员创建的。体系结构的总设计者是 Richard Stallman，他是自由软件基金会和 GNU 项目的奠基人(参见第 2 章)。

Stallman 于 20 世纪 70 年代初在麻省理工学院的人工智能实验室受训，这里与贝尔实验室的环境大不相同。其中最重要的一个区别就是麻省理工学院倾向于构建大型的、功能复杂的系统，而 Unix 程序员(只拥有很少的预算)倾向于构建简单的系统。例如，可以对比一下 Multics 和 Unix(参见第 1 章)。

尽管 Stallman 并不是典型的麻省理工学院程序员，但是他倾向于编写功能非常强大、特殊并且拥有许多神秘命令的程序。在很大程度上，您可以从 Emacs 和 **Info** 中看到这些特征。

因此，在阅读本章剩余部分以及练习使用 **Info** 时，记住要提醒自己它和 Emacs 是由同一个人创建的。如果觉得有点困惑，不必惭愧，因为其他人第一次尝试使用 **Info** 时情形也是如此。

* 与 Web 页面不同，**Info** 文件只包含纯文本以及少量的格式，没有图片。因此，在使用 **Info** 时，可以发现就像是使用早期的 Web 页面。刚开始时，Web 也是一种原始的基于文本的系统。

尽管 Info 非常复杂,它实际上只是所谓的 Texinfo 的一部分, Texinfo 是 GNU 项目的官方文档资料系统。Texinfo 是一套复杂的工具集,允许使用一个简单的信息文件生成各种不同格式的输出,这些格式包括 Info 格式、纯文本、HTML、DVI、PDF、XML 和 Docbook 等。

就我们的目的而言,我们所需知道的就是 GNU 文档资料开始时是 Texinfo 文件,然后由 Texinfo 文件生成 Info 文件。基于这一原因,有时候将 Info 称为 Texinfo。例如,如果您请教他人一个问题,他这样回答:“Have you checked with Texinfo(您查看过 Texinfo 吗)?”,那么他告诉您使用 Info。

因为 Info 非常复杂,所以我们无法在本书中包含 Info 的所有内容,而且我们也不希望这样做。接下来,我将讨论局限在 3 个主要目标上:如何使用 Info 显示希望的内容,如何操纵 Info 系统以及如何显示 Info 的帮助信息。一旦掌握了这 3 种技能,就可以在需要时进行 RTFM,从而自学需要知道的内容。

名称含义

Texinfo

当您第一次看到名称 Texinfo 时,或许会认为它应该是 Textinfo。毕竟,它是一个基于文本的信息系统的名称。实际上, Texinfo 的拼写是正确的,它来源于 TeX——一个由杰出的计算机科学家 Donald Knuth(发音为“kuh-NOOTH”)创建的排版系统。

名称 TeX 来源于希腊单词 techni,从它演变成英语单词“technical”。techni 指一种艺术、手工艺,或者更通用地说,指人们努力的最终成果。因此,字母 TeX 并不是英语字母 T-E-X,它们实际上是希腊字母 Tau、Epsilon 和 Chi,也即 techni 的前 3 个字母。如果您讲究学究式的正确性,那么您应该将 Chi 发音为苏格兰单词“loch”或者名称“Bach”中的“ch”。但是,大多数计算机人士将 Chi 发音为“K”。

那么 Texinfo 应该怎样发音呢?有 4 种选择。

第一种,如果讲究学究式的正确性,那么您应该说“Te[ch]info”,其中[ch]是上面描述的古怪发音。

如果您是一名程序员,而且希望自己看起来像一位权威人士,则可以说“Tekinfo”。

如果您希望适应非技术群体,则可以说“Texinfo”,这是大多数人从字面上读这个单词的发音。

最后,如果您希望成为自己的社交圈子中的一名领导,那么告诉每个人该名称明显应该为“Textinfo”。向他们解释第二个“t”肯定是由于偶然因素而落掉了,因此早就该修改这个错误了(实际上,您可能是正确的:即使有人发明了某些东西,也并不意味着他就有权力起一个愚蠢的名称来显摆他的聪明)*。

9.16 Info 和树

在第 8 章中讨论过数据结构的思想,数据结构是计算机科学中的一个基本概念。数据结构是一个根据一组精确的规则来存储和检索数据的实体。在那一章中,提到了最常用的

* 在我那个时代,我碰到过 Donald Knuth(他命名的 TeX)和 Richard Stallman(他命名的 GNU)。如果将他们俩关在一个房间内,或许他们能够达成共识的唯一一点就是 TeX 和 GNU 都是好名称。

数据结构包括列表、链表、关联数组、哈希表、栈、队列、双头队列(双端队列)，以及许多基于树的结构。

在第 8 章中，我们讨论了栈，因此您才可以理解 **dc** 计算器是如何处理逆波兰表示法的。在本节中，我们准备讨论树，因为它是 **Info** 用来存储和检索 **Info** 文件的数据结构。一旦理解了树，就可以搞清楚用来控制 **Info** 的命令的意思。如果不理解什么是树，当然也可以使用 **Info**，但是不会那么有趣和容易。

当计算机科学家讨论树时，指的是一族复杂的数据结构。因此，为了便于理解，我们先从一个简单的隐喻开始。

假设您决定去远足。在道路的起点有几条可以走的路，您选择其中一条。沿着这条路向前走，直到来到一个交叉路口，在这个路口上又有几条新路可供选择。再一次，您做了一个决定，选择了其中的一条路，然后一直往前走，直到又来到另一个交叉路口，被迫又进行另一个选择。依此类推。有时候，您可能走进一条死路。当出现这种情况时，您需要返回到上一个交叉路口，然后选择另一条路。

在计算机科学语言中，称每个交叉路口为一个节点。主节点(示例中道路的起点)称为根。将一个节点与另一个节点连接在一起的路径称为分支。当分支指向一条死路时，这个节点是一种特殊类型的节点，称为叶子。

下面是技术定义：对于计算机科学家来说，树就是节点、叶子和分支的集合，它们按照下述方式组织起来，即任意两个节点之间至多只有一条分支*。

尽管所有这些听起来有点复杂，但是它与我们看到的真实的树相似。作为一个例子，请看图 9-7 中所示的树。注意，与真实的树不同，计算机中的树通常将根画在顶部。

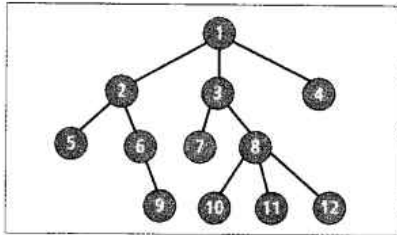


图 9-7 树

树是一种数据结构，包含许多由分支连接起来的节点。顶部(主)节点称为树的根。终端节点(终点)称为叶子。在这个例子中，节点 1 是根。节点 4、5、7、9、11 和 12 是叶子。Info 系统使用的数据结构是树，节点用来存储信息，而每个分支则是从一个节点到另一个节点的链接。

在计算机科学中，树有许多不同的类型，每种类型的树都有其自己的特征。我刚才描述的这种类型的树就是 **Info** 用来存储信息的数据结构。

* 计算机中树的思想取自数学中的图论。数学中的树与计算机的树相似，但是术语不同。

在图论中，节点称为“顶点”，而分支称为“边”。因此，如果您遇到了一名图论专家，他极有可能这样说：“Strictly speaking, the Info system uses a data structure that isn't a real tree. Since some vertices are joined by more than one edge, you should really describe it as a connected graph in which tree-like objects are embedded.”(严格地讲，Info 系统使用了一种不是真正的树的数据结构。因为一些顶点由不止一条边连接，您应该将它描述为一个连通图，图中含有一些类似于树的对象)。(现在您应该明白 Unix 人士不愿意邀请图论专家参加聚会的原因了。)

每个 Info 文件分成不同部分,并存储为一系列节点。在阅读文件时,从一个节点移动到另一个节点。这样就可以从头到尾、每次一个节点地阅读整个文件。在查看特定节点时,我们说您在访问那个节点。许多节点还包含有链接,从而可以跳到其他相关文件(就像 Web 页面上的链接一样)。使用 Info 要求具备以下 3 项基本的技能。

- (1) 使用 **info** 命令启动 Info。
- (2) 为了阅读整个文件,从一个节点移动到另一个节点。
- (3) 使用链接从一个文件跳到另一个文件。

下面按顺序讨论每一种技能。

9.17 启动 Info: info

启动 Info 系统需要使用 **info** 命令。**info** 命令的使用有两种形式。第一种,如果希望显示一条特定命令的信息,可以键入 **info**,后面跟着命令的名称。例如:

```
info date
info bc
info info
```

如果不能确定希望学习哪条命令,或者您希望浏览系统,则可以输入 **info** 命令本身:

```
info
```

当以这种方式启动 Info 时,它显示一个称为目录节点(Directory Node)的特殊节点。目录节点包含主要主题的列表,因此可以认为它是整个 Info 系统的主菜单。

9.18 学习 Info

Info 系统拥有相当数量的帮助信息,可以通过它们开始 Info 的学习。不过在开始之前,请一定要读完本章的内容。

开始这一学习旅程的地方就是 **info** 的说明书页。下面两条命令都可以完成这个任务:

```
info --help | less
man info
```

注意,在第一条命令中,单词 **help** 之前有两个连字符并且没有空格。我们将在第 10 章中讨论它的含义。

阅读了说明书页之后,就可以输入下述命令显示 Info 文件的一个简短介绍:

```
info info
```

接下来,就该阅读 Info 向导了。启动 Info 之后按 **h**(help, 帮助)键就可以进入 Info 向导。它可能有点令人迷惑,不过阅读了本章的内容之后,理解起来应该没有什么问题。

一旦阅读完 Info 向导,花一点时间显示一下 Info 的命令摘要,并浏览命令列表。按下? (问号)键就可以显示命令摘要列表。

提示

任何时候,在任何 Info 文件中,都可以显示内置的向导(通过按 **h** 键)和命令摘要列表(通过按 **?** 键)。在阅读这些文件时,不必理解所有事情。只需学习基本的命令,然后在需要时再学习所需的内容。

对于 Info 来说,没有人知道(也不需要知道)所有事情。

在使用 Info 时,有可能总要查看某个节点。如果希望离开正在阅读的节点,返回到刚才阅读的最后一个节点,可以按 **l**(字母“L”)键。

例如,假设您正在阅读包含 **date** 命令帮助的节点。您按下 **?** 键显示命令摘要列表(一个新的节点)。为了返回到 **date** 节点,所需做的全部工作就是按下 **l** 键。不要按 **q** 键,否则将完全退出 Info,返回到 shell 提示。

在阅读命令摘要列表时,您将发现 **<Ctrl>** 键使用表示法 **C-x**,而不是 **^X** 或者 **<Ctrl-X>**(这是 Emacs 的约定)。

您还将发现表示法 **M-x**。其中 **M**-代表 Meta 键,Meta 键是 Emacs 中一个非常重要的概念。现在,我只告诉您如何使用 Meta 键。有两种方法,第一种方法是在按下 **<Alt>** 键的同时按下第二个键;第二种方法是按 **<Esc>** 键,松开后再按第二个键。

例如,假如希望按 **M-x**,可以使用 **<Alt-X>** 或者按 **<Esc>** 键后再按 **<X>** 键。

9.19 阅读 Info 文件

在 Info 系统中有许多命令可以使用。图 9-8 总结了一些最重要的命令,并且接下来的几节将分别讨论它们。在阅读过程中,如果启动了 Info,跟着我们的讨论测试练习命令将会大有帮助。当阅读完本章内容之后,可以将图 9-8 作为一个参考。

每个 Info 文件都组织成一个小树,由一系列线性的节点组成。每个文件涵盖一个主要思想,例如如何使用一条特定的命令。每个节点涵盖一个主题。当开始阅读文件时,所在的位置位于表示这个文件的树的根上。在 Info 中,树的根称为顶节点。

一般而言,顶节点中包含所讨论主题的一个摘要,以及该文件中所涵盖的主题的列表。列表采用菜单的形式。

Info 文件的阅读有两种方式。第一种是按顺序阅读节点,一个接一个地从顶节点直到最后一个节点。另一种是使用菜单直接跳到特定的节点(如果希望阅读一个特定主题的话)。

阅读文件最简单的方法就是从顶节点开始,然后向下阅读整个文件。这时所需做的事情就是按 **<Space>**,从而逐屏地显示信息。当到达节点末尾时,按 **<Space>** 键将跳到树中下一个节点的开头。这样,就可以从顶节点开始,一直不停地按 **<Space>** 键来遍历整个树。

有时候需要向后移动一屏,这时可以按 **<Backspace>** 或者 **<Delete>** 键。如果位于节点的开头时按了这两个键中的一个,将跳到上一个节点(试试看)。

为了方便起见,还可以在一个节点内使用<PageDown>和<PageUp>进行移动。但是,与其他键不同,<PageDown>和<PageUp>只能在节点内移动,它们不会移动到下一个或者上一个节点中。因此,当不希望离开当前节点而向上或向下移动时,这两个键比较方便。

通用命令	
q	退出
h	启动帮助向导
?	显示命令摘要列表
阅读节点	
<PageDown>	显示下一屏
<Space>	显示下一屏
<Space>	(在节点底部时)跳转到下一个节点
<PageUp>	显示上一屏
<Backspace>	显示上一屏
<Delete>	显示上一屏
<Backspace>	(在节点顶部时)跳转到上一个节点
<Delete>	(在节点顶部时)跳转到上一个节点
在节点中移动	
b	跳转到当前节点的开头
<Up>	将光标向上移动一行
<Down>	将光标向下移动一行
<Right>	将光标向右移动一个位置
<Left>	将光标向左移动一个位置
在同一个文件中从一个节点跳转到另一个节点	
n	跳转到同一个文件中的下一个节点
p	跳转到同一个文件中的上一个节点
t	跳转到顶节点(Top Node, 文件中的第一个节点)
从一个文件跳转到另一个文件	
<Tab>	将光标移动到下一个链接上
M-<Tab>	将光标移动到上一个链接上
<Return>	到达链接指向的新节点或者文件
l	跳转到上一个(刚才观看的)节点
d	跳转到目录节点(主菜单)

图 9-8 Info 中的重要命令

此外,无论何时,当按下 **b** 键时就可以方便地跳到当前节点的开头。

最后,对于小的移动,可以使用箭头(光标)键。<Down>将光标向下移动一行; <Up>将光标向上移动一行。同理, <Right>和<Left>将光标分别向右或向左移动一个位置。

为了形象地说明 Info 节点和文件是如何连接的,请看图 9-9,该图示范了 Info 树的组

织方式。起初，该组织结构看上去可能比较复杂。但是，花点时间来理解这些连接将使 Info 的使用更加简单。

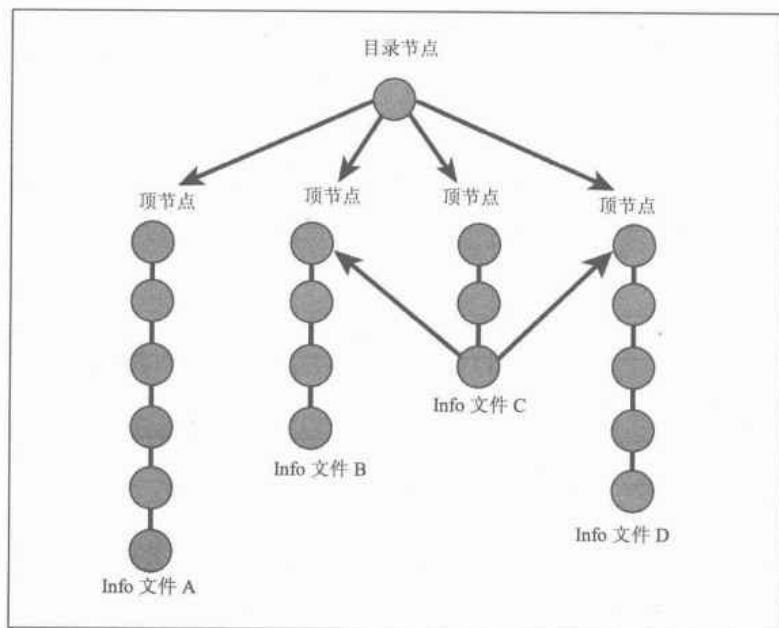


图 9-9 Info 树

这是一个高度简化的图形，示范了 Info 文件如何存储为树中的节点。在树的根部是目录节点。在我们的例子中，目录节点链接到 4 个 Info 文件，分别称为 A、B、C 和 D。每个文件存储为一串节点序列。序列中的第一个节点称为顶节点。注意文件 C 链接到文件 B 和文件 D 的顶节点。

提示

阅读 Info 文件最简单的方式就是启动 Info 文件的顶节点，然后重复地按<Space>键。通过这种方式，将会逐个节点地显示整个文件。

9.20 从一个节点跳转到另一个节点

在阅读 Info 文件时，有若干个命令可以用来从一个节点跳转到另一个节点，节点可以位于同一个文件内，也可以位于完全不同的文件中。

按 **n** 键可以跳转到当前文件中的下一个节点，按 **p** 键可以跳转到当前文件中的上一个节点，按 **t** 键则可以跳到当前文件的顶节点(开头)。

许多节点，特别是顶节点，包含一个以菜单形式显示的主题列表。每个主题实际上是一个链接，通过单击链接可以进入该主题。一些链接跳转到同一个文件内的另一个节点，另一些链接则跳转到一个完全不同的文件。

链接是可以识别的，因为它们拥有特殊的格式：一个星号(*)，后面是主题的名称，然

后是一个冒号(:)。在冒号之后,是主题的一个简短描述。有时候,您还可以看到一个资料丰富的评论。

下面举两个例子,取自前面讨论的 Info 向导。列出这两个例子就是为了示范典型菜单项的内容*:

- * Foo: Help-FOO. A node you can visit for fun.
- * Bar: Help-BAR. A second way to get to the same place.

实际链接是星号和冒号之间的菜单项(包含冒号)这一部分。它与 Web 页面上的链接相似。将光标移到链接上,按<Return>键就可以进入链接。

在链接之间移动光标的最简单方法就是按<Tab>键,这样将把光标移动到节点中的下一个链接上;或者按 **M**-<Tab>键**,这样将把光标移动到节点中的上一个链接上。另外,还可以使用箭头键(<Down>、<Up>、<Right>和<Left>)进行移动。

无论采用何种方式到达链接,一旦到达链接后,按下<Return>键就可以进行跳转。

除了单击链接进行跳转之外,还有另外两种方法可以进行跳转。正如前面所讨论的,每个 Info 文件都组织成一个简单的树,由一串节点构成。实际上,整个 Info 系统就是一棵巨大的树,每个分支都能够(直接或间接)到达系统中的每个文件和每个节点。通过按下 **d** 可以到达这棵大树的根——目录节点。因为目录节点充当整个系统的主菜单,所以 **d** 命令必须记住。

最后一种跳转的方式是按 **l**(字母“L”)键离开当前节点,返回到刚刚查看的最后一个节点。例如,假如您单击一个链接从文件 A 跳转到文件 B。此时如果按下 **l** 键,您将跳转回文件 A。

l 命令非常有用,因为重复地按该键可以沿浏览树的原路折回。该命令允许向后移动,每次一步。和 **d** 命令相同,**l** 命令也值得记住。

提示

尽管 Info 系统非常复杂,但是它只有 7 条非常重要的命令:

- (1) <Space>: 显示下一屏
- (2) <Backspace>: 显示上一屏
- (3) <Tab>: 将光标移到下一个链接
- (4) <Return>: 进入一个链接
- (5) **d**: 跳转到目录节点
- (6) **l**: 跳转到已经访问的最后一个节点
- (7) **q**: 退出

* 这两个例子直接取自 Info 向导,示范了单词“Foo”和“Bar”如何用作通用名称。参见本章前面的讨论。

** 正如前面所解释的,**M**-指 Meta 键。为了使用 **M**-<Tab>键,可以使用<Alt-Tab>,或者按<Esc>键之后再按<Tab>键。

9.21 练习

1. 复习题

1. 两个基本的 Unix 文档资料系统分别是什么？每个系统要使用什么命令访问？
2. 当阅读说明书页时，如何输入一条单独的 shell 命令？
3. 在查看 Info 节点时，使用哪条命令显示帮助向导？显示命令摘要列表？显示下一屏？跳转到当前节点的开头？跳转到下一个节点？跳转到顶节点？跳转到目录节点？

2. 应用题

1. 使用 **man** 命令显示默认 shell 的说明书页(**man sh**)。说明书页显示后，执行下述操作：显示下一屏信息；跳转到当前页的末尾；跳转到当前页的开头；搜索单词 “variable”；向前搜索同一个单词若干次；向后搜索同一个单词若干次；显示命令摘要列表；退出。

2. 如果您使用的是 Linux 或者 FreeBSD, 使用 **info** 命令显示 Info 系统本身的信息(**info info**)。显示了第一个节点之后，执行下述操作：显示下一屏信息；跳转到当前节点的末尾；跳转到当前节点的开头；跳转到顶节点；跳转到目录节点；显示命令摘要列表；退出。

3. 思考题

1. RTFM 教义要求人们在请求帮助之前自己先阅读文档资料以尝试自己解决问题。这一原则的两大优点是什么？两个缺点又是什么呢？



命令语法

在使用 Unix CLI(命令行界面)时, 需要一条接一条地输入命令。当输入命令时, 所输入的整行称为**命令行**。当在命令行的末尾按<Return>键时, 命令行的内容就发送给 shell 进行处理。

在输入命令时, 有两件事情需要了解。首先, 必须根据特定规则正确地键入命令。如何根据规则正确地输入命令是本章的主题。其次, 必须理解在 shell 处理命令的过程中发生了什么事情。这是一个重要的主题, 我们将在本书后面讨论 shell 时阐述这一主题。

Unix 中差不多有数百条命令, 而且只要您使用 Unix, 就要永不停息地学习新的命令。基于这一原因, 当需要时, 您应该能够通过联机手册自己学习。为了更好地使用联机手册, 您需要理解描述如何输入命令的形式规则。

10.1 一次输入多条命令

在大多数情况下, 一个命令行只输入一条命令。但是, 您应该知道一次输入多条命令也是可以的。在输入多条命令时, 只需用分号将各个命令分隔开即可。

下面举一个例子。您在打电话与您的朋友计划一个派对。您希望显示今天的日期以及当前月份的日历。完成这两项任务的命令分别为 **date** 和 **cal**(参见第 8 章)。当然, 您可以分开输入这两条命令:

```
date
cal
```

但是, 使用一个分号作为命令分隔符就可以在一行上输入这两条命令:

```
date; cal
```

注意命令行的末尾并不需要分号。

下面再举一个例子。这个例子使用 **cp** 命令复制文件(参见第 25 章), **ls** 命令显示文件的信息(参见第 24 章), 以及 **rm** 命令删除(移除)文件(参见第 25 章)。

您的目标是为一个名为 **data** 的文件生成两份副本。两份副本分别称为 **backup1** 和

backup2。在生成文件副本之后，您希望删除原始文件。然后，列出所有的文件，看看拥有什么文件。相关命令如下所示：

```
cp data backup1
cp data backup2
rm data
ls
```

不要担心细节问题，我们将在本书后面讨论这些命令。这里的重点就是，我们不是在 4 个独立的行上键入这些命令，而是通过使用分号分隔这些命令，在一行上输入它们：

```
cp data backup1; cp data backup2; rm data; ls
```

您或许会问，在一行上输入所有命令有什么意义呢？毕竟，这只是以另一种方式键入相同数量的字符而已。

答案在于，从长远来看，如果您能够一次考虑好接下来好几步的行动，就像下象棋者在决定下一步棋如何走时会考虑接下来的各种变化一样，那么您就可以更有效地使用 Unix。尽管目前看上去这似乎不大现实，但是在您更有经验时这一点就会十分重要。

从第 15 章至第 19 章，我们将展示如何将各种命令和工具组合在一起，从而形成复杂的命令行。为了更好地完成这一点，您需要开发自己的技能，成为一名 Unix 问题解决者。现在，我希望您通过查看有多少机会可以在同一命令行上输入多条命令，开始以这种方式进行思考。

提示

一名优秀的 Unix 用户和一名伟大的 Unix 用户之间的区别在于解决问题的能力，即当问题出现时，能够将各种工具和思想快速、创造性地组合在一起，并且还不忙乱。实际上，一些人认为这是熟练使用 Unix 的本质要求。

10.2 输入命令时会发生什么事情

在输入命令时，需要键入命令的名称，后面还可能跟一些其他信息。例如，考虑下面的 **ls** 命令行：

```
ls -l -F file1
```

该命令有 4 部分：命令的名称 **ls**，后面跟着 **-l**、**-F** 和 **file1**（稍后将更具体地讨论其他部分的称呼）。

当按下 <Return> 键时，shell 就会处理命令。实际细节有点复杂，我们将把讨论留在后面的各章中。现在先提供一个简化的描述，该描述适用于大多数人们。

shell 通过假定命令行的第一部分是希望运行的命令的名称来处理命令。然后 shell 根据该名称搜索并执行程序。例如，如果输入的是上面的 **ls** 命令，那么 shell 将查找并运行 **ls** 程序。

如果 shell 找不到输入的命令的名称, 那么 shell 将显示一个错误消息。例如, 假设输入了:

```
harley
```

虽说难以相信, 但是 Unix 并没有 **harley** 这条命令, 所以您将看到:

```
harley: not found
```

当 shell 查找到希望运行的程序时, 它就会运行这个程序。这时, shell 将整个命令行的一份副本发送给该程序。然后轮到程序来决定如何处理所有的信息了。在上面的例子中, **ls** 程序将决定如何处理 **-l**、**-F** 和 **file1**。

10.3 命令语法

Unix 命令必须根据一个明确定义的规则键入。一个标点符号放错位置或者单词拼写错误都会使整个命令无效。大多数时候, 系统会显示一个错误消息, 并且忽略这个命令。但是, 在最坏的情况下, 一条错误键入的命令可能会错误地执行, 造成其他问题。因此, 学习如何正确地输入命令十分重要。关于如何输入命令的格式描述称为命令语法(command syntax), 不正式的称法可以称之为语法。

如何学习命令的语法呢? 很简单。一旦理解了通用规则和约定, 就可以通过查找手册或者 Info 系统(参见第 9 章)自己学习如何使用任何命令。

如果您是一名初学者, 那么您应该知道有许多命令现在超出了自己的能力范围。实际上, 有一些命令您可能永远不会理解(或者永远不需要理解)。无论如何, 我的目标是当您阅读完本章内容之后, 应该能够理解任何命令的语法, 即使您不理解命令本身*。

通常, Unix 命令语法可以表示为: 键入命令的名称, 后面是“选项(options)”, 再后面是“参数(arguments)”:

命令名称 选项 参数

选项用于设置命令执行任务的方式, 而参数指定命令使用的数据。

考虑下面的例子:

```
ls -l -F file1 file2 file3
```

在这个例子中, 命令名称是 **ls**。选项是 **-l** 和 **-F**。参数是 **file1**、**file2** 和 **file3**。

下面, 我们开始讨论细节。

* 这并没有听起来那么奇怪。例如, 当您学习一门外语时, 用不了多久就能够阅读自己不能理解的句子。对 Unix 来说也是如此。当结束本章内容时, 您就能够理解任何 Unix 命令的语法。但是, 除非您拥有丰富的经验, 否则大多数命令对您来说依然神秘。

我的目标是确保您对基本的原理(例如命令语法)有一个坚实的掌握, 从而能够通过定期练习及花点时间自学弥补差距。例如, 无论何时, 当您有问题时, 我鼓励您花点时间使用手册或者 Web 自己寻找答案(如果这听起来对您十分愉快, 那么您是一名十分幸运的人)。

10.4 选项

当键入命令时, 可以使用选项来修改命令执行任务的方式。当阅读 Unix 文档资料时, 有时候会看到称选项为开关(Switches)或者标志(Flags), 因此您也要知道这些单词。

顾名思义, 选项是可选的。大多数命令都至少拥有几个选项, 有一些命令的选项如此之多, 您甚至可能会怀疑有没有人能够全都知道这些选项。答案是没有人能够这样。即使是有经验的人也要依赖于手册来查找模糊的选项。

在命令行中, 选项直接位于命令名称之后。选项通常由一个连字符后跟一个字母, 或者两个连字符后跟一个单词组成。例如:

```
ls -l -F file1 file2 file3
ls --help
```

在第一条命令中, 有两个选项 **-l** 和 **-F**。在第二条命令中, 有一个选项 **--help**。

选项的任务就是控制命令的动作。例如, 在上面的例子中, 选项 **-l** 告诉 **ls** 命令显示“长”列表。通常, **ls** 命令只显示文件的名称。当使用 **-l** 选项时, **ls** 将列举每个文件的额外信息, 以及文件的名称。

有时候, 您可能看到选项是一个数字, 例如:

```
ls -l file1 file2 file3
```

您必须要小心。例如, 在这个例子中, 不要混淆了 **-l**(小写字母“l”)和 **-1**(数字 1)。

当使用多个单字符的选项时, 可以将它们组合在一起, 只使用一个连字符。此外, 选项可以以任意的顺序指定。因此, 下面所有的命令都是等价的:

```
ls -l -F file1
ls -F -l file1
ls -lF file1
ls -Fl file1
```

对于所有的 Unix 命令, 都必须确保正确使用了大写字母和小写字母。例如, 命令 **ls** 既拥有 **-F** 选项, 也拥有 **-f** 选项, 这两个选项完全不同。通常情况下, 大多数选项都是小写字母(正如第 4 章中所说, Unix 中的几乎所有内容都是小写的)。

提示

当在手册中查找命令时, 将会看到每个选项都对应一个解释。但是, 说明书页并没有说明单字符选项可以组合在一起。例如, **-l -F** 可以组合成 **-lF**。这是基本的 Unix 文化, 手册假定您已经知道这一点。

此时, 出现了一个有趣的问题: 当谈论选项时, 连字符如何发音呢?

以前, 它发音为“minus”, 这或许是因为它比“hyphen”容易表达。例如, 以前的人将 **ls -l** 发音为“L-S-minus-L”。

但是, 在 Linux 文化中, 将连字符称为“dash”比较常见, 特别是在年轻用户中。因此, Linux 人士通常将 **ls -l** 发音为“L-S-dash-L”。同样, 将“**ls --help**”发音为

“L-S-dash-dash-HELP”。

在键入命令时，“minus”和“dash”之间的区别并不重要。但是，在谈话时，大声说出来的东西有重要的社会含意。

例如，假设您的教师在课堂上叫您，并问道：“您能说明 **ls** 命令如何显示长文件列表吗？”对于大多数学生来说，答案是：“使用 **dash-L** 选项。”但是，如果这位教师是在 20 世纪 90 年代中期之前学的 Unix，那么为了显示对一位老人的尊敬，应该说：“使用 **minus-L** 选项。”（记住，有一天您也会老的。）

10.5 -选项和--选项

当谈论选项时，单个的-格式是比较古老的一种。它可以追溯到最早的 Unix 系统(参见第 2 章)，那时程序员期望简化命令行。许多命令的名称只有两个或者三个字母，而且所有的选项只有一个字母。

许多年过去之后，当 GNU 实用工具开发出来时(参见第 2 章)，设计人员希望能够使用较长的选项。但是，长时间的约定认为所有选项应该只有一个字母，而且选项可以按前面解释的方式组合在一起。例如，假如 **shell** 遇到了下述命令，会发生什么事情呢？

```
ls -help
```

因为多个选项可以组合在一起，所以 **-help** 选项将被解释为 4 个单独的选项：**-h**、**-e**、**-l** 和 **-p**。

当然，规则也可以修改，但是这是一个极端的修改，将使许多已有的程序无效，并且强制太多的人改变他们的习惯。实际上，人们最后决定允许使用更长的选项，但是它们前面要前置两个连字符，而不是一个。通过这种方式，系统可以进行扩展，且不必损害已有的系统。

因为 Linux 使用 GNU 实用工具，所以如果您是一名 Linux 用户的话，就可以看到两种类型的选项。对于其他大多数系统来说，通常只有单字符的选项。

设计--选项的主要目的是使用较长的选项，从而使选项更容易理解和记忆。另一方面，较长选项键入速度较慢，并且更易于拼错。基于这一原因，许多命令同时提供了长的和短的选项。用户可以根据自己的情况选择使用。

例如，命令 **ls**(列举文件信息)拥有一个 **-r** 选项，该选项使 **ls** 命令以相反的顺序列举文件。对于 Linux 来说，该命令还有一个 **--reverse** 选项，与 **-r** 选项的作用相同。因此，下述两条命令是等价的：

```
ls -r
ls --reverse
```

在--选项领域中，当使用 GNU 实用工具时经常会遇上两个选项。如果您是 Linux 用户，那么这两个选项值得记住。

第一个常见的--选项是 **--help** 选项，如果使用该选项，将显示命令的语法摘要。通常，

这个选项单独使用。例如：

```
ls --help
date --help
cp --help
```

在许多情况下，摘要可能很长，以至于在阅读之前就滚动出屏幕。如果发生这种情况，可以将输出发送给 **less** 程序，后者将每次一屏地显示输出信息。实现这种操作时，只需键入一个|(竖线)字符，后面跟着 **less** 即可。例如：

```
ls --help | less
date --help | less
cp --help | less
```

有关 **less** 的讨论请参见第 21 章。

第二个常见的--选项是--version:

```
ls --version
date --version
cp --version
```

该选项显示系统上所安装程序的版本信息。通常，这并不是有价值的信息。但是，如果遇到了问题，那么它有时候可以帮助了解正在运行程序的版本。

提示

键入选项的规则对几乎所有的 Unix 命令而言都是相同的：每个选项或者以一个连字符，或者以两个连字符开头；单字符的选项可以组合起来。但是，也有几个例外情况。

有一些命令使用没有连字符的选项。有些命令接受连字符，但是并不要求。最后，还有一些命令不允许组合单字符选项。

幸运的是，这些例外十分罕见。当然，如果您在使用某个命令时遇到了问题，可以查看说明书页。说明书页是最终的权威参考。

10.6 参数

前面提到过 Unix 命令的通用语法可以表示为命令的名称，后面是选项，再后面是参数：

命名名称 选项 参数

前面已经讨论过命令的名称和选项，因此下面讨论如何使用参数。

参数在命令行上用来向希望运行的程序传递信息。考虑下面的例子，该例在前面讨论选项的过程中使用过：

```
ls -l -F file1 file2 file3
```

在这个命令行中，命令是 **ls**，选项是 **-l** 和 **-F**，而参数是 **file1**、**file2** 和 **file3**。

参数的含义随着命令的不同而不同。典型情况下，参数指定程序执行动作所需的数据。在我们的例子中，指定了 3 个文件的名称。

考虑另一个例子：

man date

在这个例子中，命令名称是 **man**，没有选项，而参数是 **date**。在这条命令中，我们告诉 **man** 显示 **date** 的说明书页。

最后一个例子：

passwd weedy

这里使用的命令是 **passwd**(change password, 改变口令)，该命令有一个参数 **weedy**。在这个例子中，我们表示希望改变用户标识 **weedy** 的口令(参见第 4 章)。

名称含义

参数

Unix 命令的一般语法是：

命令名称 选项 参数

单词“option(选项)”有意义是因为命令行选项允许您选择命令的执行方式。但是，单词“argument(参数)”呢？很明显，它并不表示某项智能活动，或者“准备建立一个命题的一系列相互连接的陈述”^{*}。

实际上，计算机术语“参数”取自数学，在数学中它指的是自变量。例如，在方程 $f(x) = 5x + 17$ 中，参数是 x 。从这种意义上讲，参数就是命令或者函数所操作的对象。

在英语中，单词“argument”取自拉丁语中的单词 *arguere*，它的含义是阐明或者解释。

10.7 空白符

在输入命令时，必须要确保将每个选项及参数分开。为了实现该目的，命令的每个部分之间必须至少有一个空格或者制表符。例如，下面是输入同一条命令的几种方式。注意这里显式指定了使用 **<Space>** 和 **<Tab>** 键的地方：

```
ls<Space>-l<Space>-F<Space>file1
ls<Tab>-l<Tab>-F<Tab>file1
ls<Space><Tab>-l<Space>-F<Tab><Tab><Tab>file1
```

^{*} 如果您没有辨别出这一引用出自 Monty Python 的歌曲“The Argument Clinic”，那么您在文化培训方面还存在极大的不足。我的建议是立即花一些时间来弥补这一缺陷(提示：使用 Web 来搜索“monty python”“argument clinic”)。

当然，一般情况下命令的每个部分之间使用一个空格即可。实际上，对于一些 shell 来说，只能使用空格，因为制表符拥有一个特殊的功能(称为命令自动补全)。但是，使用空格和制表符作为分隔符的思想如此重要，所以它们拥有自己的名称：空白符。

在使用 Unix 的过程中，这一思想会反复出现，因此下面给出一个明确的定义。在命令行中，空白符指的是一个或者多个连续的空格或者(对于一些 shell 来说)制表符。在其他情形中，空白符可能指一个或者多个连续的空格、制表符或者新行字符(有关新行字符的讨论，请参见第 4 章)。

如果您是一名 Windows 或者 Macintosh 用户，就可以看到含有空格的文件名称。例如，Windows 系统就使用诸如“Program Files”、“My Documents”等文件夹(目录)名称。在 Unix 中，认为命令行中的空格是空白符，这样就不能在文件的名称中使用空格。同理，Unix 命令的名称中也没有空格。

提示

当选择拥有不止一个部分的名称时，永远也不要再在名称的中间使用空格。不过，可以使用连字符(-)或者下划线(_)。

例如，下面都是有效的名称：

```
program-files
program_files
my-documents
my_documents
```

我个人喜欢用连字符，因为连字符比下划线更容易阅读。

(提醒：Unix 区分大写字母和小写字母。另外，在大多数时候，我们只为名称使用小写字母。)

名称含义

空白符

术语“whitespace，空白符”指连续的空格或者制表符，空白符用来将两个项分开。该名称起源于最初在纸上进行打印的 Unix 终端。在键入命令时，各个单词之间真的留有空白。

Unix shell(命令处理器)被设计得非常灵活，它对命令行各部分之间有多少个空格并不介意，只要词与词之间分开就可以。因此，术语“空白符”意味着任意数量的空格和制表符。

后来，对于特定的应用程序，该术语的含义扩展为任意数量的空格、制表符或者新行字符(第 4 章中讲过，新行字符是在按下<Return>键时生成的字符)。

10.8 一个或多个；零个或多个

下一节将讨论命令的标准描述方法。但是，在此之前，需要先定义两个重要的短语：

“一个或多个”和“零个或多个”。

当看到短语“一个或多个”时，它意味着必须至少使用一个。下面举例说明。

在第9章中，讨论了如何使用 **whatis** 命令。根据联机手册中关于某命令的条目，**whatis** 命令显示一条命令的简要描述。当使用 **whatis** 时，必须指定一个或者多个命令名称作为参数。例如：

```
whatis man cp
whatis man cp rm mv
```

第一个例子有 2 个参数；第二个例子有 4 个参数。因为该命令的规范要求“一个或多个”名称，所以必须至少包含一个名称——它并不是可选的。

另一方面，“零个或多个”意味着可以使用一个或者多个表示，但是也可以一个都不使用。

例如，前面提到过的 **ls** 命令加上 **-l** 选项可以列出指定文件的信息。该命令的准确格式要求指定零个或多个文件名称。下面示范 3 个例子：

```
ls -l
ls -l file1
ls -l file1 file2 data1 data2
```

当看到要求零个或多个的规范时，您应该问：“如果我一个也不用会怎么样？”一般来说，这种情况下，系统使用一个默认值，即假定值。

对于 **ls** 命令来说，默认值是当前“工作目录”中的所有文件。这样，如果像第一个例子那样，不指定任何文件名称，那么 **ls** 将列出当前工作目录中所有文件的信息。当指定一个或者多个文件名称时，**ls** 将只显示这些特定文件的信息。

提示

每当被告知可以使用零个或多个对象时，都要问一问：“默认值是什么？”

10.9 命令的形式描述：语法

学习新命令的一种好方法是回答下述 3 个问题：

- 命令是做什么的？
- 如何使用选项？
- 如何使用参数？

通过使用 **man** 命令查阅联机手册(参见第9章)，然后阅读该命令的摘要就可以知道该命令的用途。如果需要了解更多的信息，可以查阅该命令的完整描述。

在查阅说明书页时，能见到命令的语法：命令的准确、正规的使用规范。通俗一点讲，可以认为语法是命令如何使用的“官方”描述。

在 Unix 中，命令语法遵循 7 条规则。其中前 5 条规则是最基本的规则，因此我们从这 5 条规则开始。稍后再讨论另外 2 条规则。

- (1) 方括号中的项是可选的。
- (2) 不在方括号中的项是必选项，必须作为命令的一部分输入。
- (3) 黑体字必须按原样准确键入。
- (4) 斜体字必须用适当的值代替。
- (5) 后面接省略号(...)的参数可以重复任意多次。

现在举例说明这些规则的应用。下面是某个特定 Unix 系统中 **ls** 命令的语法：

```
ls [-aAcCdFfGgILlQqRrstul] [filename...]
```

从这个语法描述中，可以得出什么结论呢？

- 该命令有 18 个不同的选项。可以使用 **-a**、**-A**、**-c**、**-C** 等选项。因为选项是可选的，所以用方括号将它们括起来。换句话说，就是可以使用零个或多个选项。
- 该命令有一个参数 *filename*。该参数是可选的，因为它也是用方括号括起来的。
- 该命令的名称和选项都是黑体字印刷。这意味着必须准确无误地键入它们。
- 参数是斜体字，这意味着必须用适当的值代替它(在本例中，指的是文件名或者目录名)。
- 参数后面接“...”，这意味着可以使用不止一个参数(指定不止一个文件名)。因为参数本身是可选的，所以要更准确的话，可以说必须指定零个或多个文件名。

基于这条语法，下面列举了一些有效的 **ls** 命令。记住，单连字符选项既可以单独键入，也可以组合在一起使用一个单独的连字符：

```
ls -l
ls file1
ls file1 file2 file3 file4 file5
ls -Fl file1 file2
ls -F -l file1 file2
```

下面是一些无效的 **ls** 命令。第一个命令是无效的，因为它使用了一个未知的选项(**-z**)：

```
ls -lz file1 file2
```

接下来的命令也是无效的，因为选项位于参数之后：

```
ls file1 -l file2
```

该例非常典型地说明了为什么必须严格遵循命令的语法。

通常，选项必须位于参数之前(不过也有例外，稍后将介绍)。因为单词 **file1** 没有以连字符开头，所以 **ls** 假定它是一个参数，并且假定它后面的都是参数。因此，**ls** 认为指定了 3 个文件的名称，即 **file1**、**-l** 和 **file2**。当然，当前目录中没有名为 **-l** 的文件，所以命令的结果将不是我们所希望的。

最后两个语法针对更复杂的情形。

- (6) 如果一个单独的选项和一个参数组合在一起，那么该选项和参数必须同时使用。

下面的例子(一个 Linux 版 **man** 命令的简化版本)描述了这一规则：


```
man [-P pager] [-S sectionlist] name...
```

在这个例子中, 如果希望使用 **-P** 选项, 必须在该选项之后立即接上参数 *pager*。同样, 如果希望使用 **-S** 选项, 必须在该选项之后立即接上参数 *sectionlist*。

可以看出, 这种类型的语法是“所有选项都必须位于参数之前”这一通用指导原则的例外。在这个例子中, 参数 *pager* 位于选项 **-S** 之前是有可能的。

下面是最后一个语法规则。

(7) 由|(竖线)字符分开的两个或多个项, 表示可以从这个列表中选择一个项。

下面的例子描述这一规则, 这个例子示范了一个 Linux 版本的 **who** 命令的语法:

```
who [-abdHilmpqrstTu] [file | arg1 arg2]
```

该语法说明该命令可以使用一个名为 *file* 的参数, 或者两个名为 *arg1* 和 *arg2* 的参数。下面列举该命令的两个例子。其中, 第一个例子指定 *file*; 第二个例子指定 *arg1* 和 *arg2*:

```
who /var/run/utmp
who am i
```

现在还不用关心细节^{*}。我希望您注意的是当拥有不止一个选项时, 如何使用竖线来描述它们。

10.10 使用 Unix 手册学习命令语法

当阅读印刷资料(例如书籍)时, 很容易看出哪些单词是黑体字, 哪些单词是斜体字。但是, 当在显示器上查阅联机手册时, 可能看不到特殊的字体。

在一些系统上, 有黑体字和斜体字; 在另一些系统上, 没有黑体字和斜体字。在不显示斜体字的系统上, 通常使用下划线代替斜体。无论如何, 您必须适应自己使用的特定系统, 从而可以根据上下文推断哪些单词是参数。通常这并不困难。

典型的说明书页将解释每一个可能的选项和参数。但是, 作为一种语法摘要, 一些版本的手册使用简化的形式, 不会列出单个的选项, 而是使用单词“options”表示。下面举

^{*} 如果阅读 **who** 命令的说明书页, 就会明白第一个参数(*file*)是文件的位置, **who** 从这个文件中提取信息。您可以指定自己的文件, 以替代默认的文件。可能您平时并不这样做, 但是该思想却简单易懂。

另两个参数(*arg1* 和 *arg2*)非常有趣。第 8 章曾经讲过可以使用下述命令显示当前登录到系统中的用户标识:

```
who am i
```

更有趣的是, 不管键入的是什么内容, 只要指定两个参数, **who** 命令都以这种方式响应。例如, 可以键入:

```
who are you
who goes there
```

或者, 如果您足够勇敢, 还可以键入:

```
who is god
```

您认为命令为什么会以这种方式编写呢?

一个例子，之前我们采用下述方式显示 **ls** 命令的语法：

```
ls [-aAcCdfFgilLqrRstu1] [filename...]
```

使用简化形式后，语法将变成：

```
ls [options] [filename...]
```

不用担心，无论在语法摘要中是否能看到选项，在详细的命令描述中都将枚举并解释各个选项。

10.11 如何学习众多的选项

在前面使用的例子中，**ls** 命令有 18 个不同的选项。实际上，一些版本的 **ls** 命令甚至拥有更多的选项，其中许多选项还是所谓的--选项。这里出现了一个问题，如何学习如此众多的选项呢？

答案是不必学习。没有人能够记住每个命令的全部选项，即使是他们经常使用的命令。最好的方法是只记住最重要的选项。当需要使用其他选项时，可以查阅其他选项的使用方法，而联机手册就是为此设计的。

Unix 程序员的特征之一就是他们倾向于编写拥有许多选项的程序，而其中有许多选项您大可不必理会。此外，不同版本的 Unix 也会对同一个命令提供不同的选项，这种现象还并不罕见。

前面使用过的 **ls** 命令取自一个特定类型的 Unix。其他系统的 **ls** 命令选项与此不尽相同。但是，最重要的选项——也就是使用最多的选项，不会由于系统的不同而变化太大。

在本书中，将解释许多 Unix 命令。在解释 Unix 命令的过程中，将只对那些最重要的选项和参数(也就是那些经常需要使用的)给予描述。如果您正在学习一条命令，而且想了解该命令所有可能的选项和参数，那么您可以查阅系统的联机手册。

考虑第 9 章中曾经描述过的 **man** 命令的语法：

```
man [section] name...
```

```
man -f name...
```

```
man -k keyword...
```

由于该命令可以以 3 种不同的方式使用，所以很容易使用 3 种不同的描述说明该命令的语法。

第一种使用 **man** 的方式是一个可选的 *section* 数字及一个或多个 *name* 值。第二种方式是使用 **-f** 选项及一个或多个 *name* 值。第三种方式是使用 **-k** 选项及一个或多个 *keyword* 值。

这些并不是 **man** 仅有的选项，只是 **man** 命令最重要的选项。在一些系统上，**man** 拥有大量的选项。但是，对于日常工作来说，**-f** 和 **-k** 是唯一需要使用的选项。如果希望知道自己的系统上 **man** 命令还有什么选项，可以查阅手册。

为了结束本章的讨论，下面再举最后一个例子。正如第 9 章所解释的，可以使用 **whatis** 命令代替 **man -f**，使用 **apropos** 命令代替 **man -k**。这两个命令的语法如下所示：

```
whatis name...
apropos keyword...
```

该语法表明, 在使用其中任一命令时, 要输入命令名称(**whatis** 或 **apropos**), 后跟一个或多个参数。

提示

一些命令在各个版本的 Unix 之间有一些微小的差别。本书一般使用 GNU/Linux 版本的命令。大多数时候, 这没有什么问题, 因为最重要的选项和参数在大多数类型的 Unix 中都是相同的。

但是, 如果遇到了问题, 最终了解程序是否可以在您的系统上运行所依靠的资源是联机手册, 不是本书(或者其他任何书籍)。

10.12 练习

1. 复习题

1. 如何在同一行上输入多个命令?
2. Unix 命令的语法可以表示为: 命令名称 选项 参数。那么什么是选项呢? 什么是参数呢?
3. 什么是-选项? 什么又是--选项? 每一种选项各有什么用途?
4. 什么是空白符?
5. 当学习一个新程序的语法时, 应该询问哪 3 个基本问题?

2. 应用题

1. 人们经常需要在同一行上输入多个命令。通过在一行上键入下述 3 条命令, 为系统的状态创建一个简短的摘要: **date**(时间和日期)、**users**(当前登录系统的用户标识)和 **uptime**(系统已经启动多久了)。
2. 为下述程序编写语法。命令名称是 **foobar**。程序有 3 个选项: **-a**、**-b** 和 **-c**。其中 **-c** 选项有一个可选的参数 **value**。最后, 程序必须有一个或多个 **file** 参数的实例。

3. 思考题

1. 许多 GNU 实用工具(Linux 和 FreeBSD 使用)同时支持传统的-(连字符)选项, 以及较长的--(双连字符)选项。为什么 GNU 开发人员觉得有必要引入一种新类型的选项呢? 新类型的选项有什么优点? 其缺点又是什么?

shell

众所周知, shell 是读取并解释命令的程序。从一开始, shell 就被设计成一个常规的程序, 一个执行任务时不要求特殊权限的程序。从这种意义上讲, shell 就像 Unix 系统中运行的其他任何程序。

因为这一基本的设计, 所以任何拥有编程必备技能的人都可以设计自己的 shell, 然后与其他 Unix 用户分享。多年以来, 事情就是这样发生的, 因此现在有许多的 shell 可供使用。Unix 系统通常至少拥有其中的几种, 用户可以根据自己的希望任意选择一种使用, 甚至还可以在 shell 之间来回切换。如果想尝试一个自己的系统上还没有安装的 shell, 则可以从 Internet 上免费下载, Internet 上提供了大量的免费 shell。

在本章中, 将回答下面几个问题: 什么是 shell? 为什么 shell 如此重要? 最流行的 shell 有哪些? 应该选择使用哪个 shell? 在接下来的几章中, 将讲授最重要的几个 shell 的使用方法, 包括: Bash、Korn shell、C-Shell 和 Tcsh。

11.1 什么是 shell

从开始使用 Unix 起, 您就会听到许多关于 shell 的话题。那么这个“shell”到底是什么呢? 答案有若干种。

最短的技术回答就是 shell 是一个程序, 充当用户界面和脚本解释器, 允许用户输入命令以及间接地访问内核的服务。

下面再从不太技术的角度出发, 给 shell 一个更详细的、包含两部分的描述。首先, shell 是一个**命令处理器**: 一个读取并解释所输入命令的程序。每键入一条 Unix 命令, shell 就读取该命令, 并指出应该怎样做。大多数 shell 还提供了一些工具, 使日常工作更加便利。例如, shell 允许重新调用、编辑以及重新输入前面的命令。

其次, shell 还支持一些类型的编程语言。使用该语言, 可以编写由 shell 解释的程序, 这些程序称为 **shell 脚本**。这些脚本可以包含常规的 Unix 命令, 以及特殊的 shell 编程命令。每种类型的 shell 都拥有自己特定的编程语言和规则。但是, 位于同一“家族”之内的 shell 通常使用相似的编程语言(在本章后面将讨论两个主要的 shell 家族)。

但是, 这些解释没有一个真正地捕捉到 shell 的本质。您知道, shell 是使用 Unix 的主

要界面。因为 shell 有许多种，所以可以选择使用哪种类型的界面，但是所选择的 shell 将影响您对 Unix 的感受。

可以想象，由于有许多种 shell，因此到底哪些 shell 是最好的，到底应该尽量避免使用哪些 shell，即便是在专家之间，意见也大相径庭。但是，除非您是一名有经验的 Unix 用户，否则使用哪种 shell 无关紧要。各种 shell 之间的区别尽管重要，但是对于初学者来说并没有什么意义：初学者最好使用系统上的默认 shell。一旦您有经验了，就可以选择使用一个自己最喜欢的 shell，然后使用它为自己创建一个高度定制的工作环境。

此时，一旦知道了如何自如地操纵自己的工作环境，您将开始理解人们对 shell 的神秘感。虽然您既看不到它也摸不到它，但它一直等在那里，等待着按照您个人的思考过程为您的每个需求提供服务(如果您是一名泛神论者，那么所有这些都会让您感觉非常奇妙)。

提示

精通正在使用的 shell 远比花大量时间试图选择一个“正确”的 shell，或者试图劝说忙碌的系统管理员在您的系统上安装一个新 shell 更为重要。

“如果不能使用自己喜欢的 shell，那么就喜欢正在使用的 shell。”——Harley Hahn

名称含义

shell

对于“shell”这一名称有 3 种理解方式。第一种，Unix 的 shell 提供了一种定义明确的界面，用来保护操作系统的内部。从这种意义上讲，shell 的作用就像牡蛎的壳，保护其脆弱的部分免受外界的伤害。

第二种，可以将 shell 想象成海里的一种贝壳，其外壳一圈一圈地向上旋转形成一个螺旋。在使用 Unix 的 shell 时，随时可以暂停正在做的工作，启动另一个 shell 或者另一个程序。因此，可以启动任意数量的程序，并且每个程序都位于它的前辈的“内部”，就像贝壳的螺旋一样。

但是，我的建议是要忍住不让自己问这个问题，即名称“shell”的含义是什么。相反，要将单词“shell”看成一个全新的技术术语(就像 RTFM 或 foo 一样)，并且让自己随着使用 Unix 的经验日益增加而逐渐领会它的含义。

11.2 Bourne Shell 家族：sh、ksh、bash

shell 是程序，并且像所有程序一样，它由运行它时所键入的命令的名称来标识。最早的 shell 是在 1971 年由 Unix 的创建者 Ken Thompson 开发的，即在 Unix 诞生两年后开发的(有关 Unix 的历史请参见第 2 章)。为了与给程序命名短名称这一传统一致，Thompson 将该 shell 命名为 **sh**。

假设您是 20 世纪 70 年代初期的一名早期用户。下面是您使用 **sh** 程序的方式。首先，键入用户标识和口令登录系统(参见第 4 章)。一旦口令通过了验证，就开始执行各种启动过程，然后 Unix 为您运行 **sh** 程序，从而开始工作会话。

sh 程序显示一个 shell 提示(参见第 4 章)并等待输入命令。每输入一条命令, **sh** 就执行所需的工作处理命令。一旦命令处理完毕, **sh** 就显示一个新的 shell 提示, 并等待输入另一条命令。最终, 您通过按下 **^D** 发送一个 **eof**(end of file, 文件结束)信号(参见第 7 章)告诉 **sh** 程序不再有输入数据了。在捕获到这个信号(参见第 7 章)后, **sh** 程序终止, 并将您注销, 结束工作会话。

现在, 使用 shell 的过程基本上与 1971 年时相同。固然, 现代的 shell 相比原始的 **sh** 程序, 功能要强大许多, 但是它们仍然充当界面, 一条接一条地读取命令, 并且在没有数据时终止(这就是为什么将 shell 称为命令行界面的原因)。

第一个 shell——可以称它为 Thompson shell, 于 1971 年至 1975 年使用, 随第一版 Unix (Version 1) 至第六版 Unix (Version 6) 一起发行。1975 年, 贝尔实验室的一群程序员(由 John Mashey 领头)编写了一个新的 shell。该新 shell 作为一个特殊版本的 Unix (称为 Programmer's Workbench 或者 PWB) 的一部分于 1976 年发行。因为 Mashey Shell (也叫 PWB Shell) 被设计用来替代原始的 shell, 所以也将它命名为 **sh**。

为一个新 shell 赋予相同的名称的优点在于, 对于用户来说, 当引进新 shell 时不必做任何特殊事情。一天, 您运行 **sh** 程序, 得到的是旧的 shell; 第二天, 还是运行 **sh** 程序, 得到的是新的 shell。只要新 shell 与旧 shell 兼容, 则一切正常。您可以做在旧 shell 中做的任何事情, 而且如果您愿意, 还可以使用新 shell 的增强特性。

当新程序以这种方式与旧程序相关时, 我们就可以说新程序向后兼容旧程序。例如, Mashey Shell 就向后兼容原始的 Thompson Shell。

因为 shell 被设计为常规的程序, 所以任何拥有足够专业知识的人都可以根据自己的爱好修改已有的 shell, 甚至编写自己的 shell^{*}。1976 年, 另一名贝尔实验室的程序员 Steve Bourne 着手开发一种全新的 shell。因为所开发的新 shell 准备替换以前的 Thompson Shell, 所以 Bourne Shell 也命名为 **sh**。

不管是 Mashey 的 shell 还是 Bourne 的 shell 都提供了许多重要的改进, 特别是在编程方面, 因此在很短的时间内, 两种 shell 就在贝尔实验室获得了众多的支持者。但是, 尽管它们都向后兼容已有的 **sh** 程序, 但是它们彼此之间不兼容。这样就导致了一个内部争论, 即哪种 shell 应该成为标准的 Unix shell。该争论的结果非常重要, 因为它改变了未来几年 Unix 的发展进程。

在连续 3 届 Unix 用户大会上, Mashey 和 Bourne 都在讨论他们各自的 shell。在大会间隔期间, 他们都努力增强他们的 shell, 添加新的特性。为了永久地解决这个问题, 人们专门成立了一个委员会来研究这一问题。最终他们选择的是 Bourne 的 shell。

因此, 在 Unix 发行第 7 版 (Version 7) 时, Bourne shell 成为所有 Unix 用户的默认 shell。实际上, Bourne shell 非常稳定, 设计特别优秀, 多年以来它一直是 Unix 的标准 shell。有时候, 还会有新版本的 Bourne shell 发布, 不过这些新 shell 都保持向后兼容, 并且都命名为 **sh**。现在, Bourne shell 获得了广泛的应用, 所有兼容的 shell, 不管是旧的还是新的, 都被认为是 Bourne Shell 家族的成员。

^{*} John Mashey 曾经向我描述那段在贝尔实验室的时光。他说: “那时, 我们的部门有 30 来个人, 大概有 10 种不同风格的 shell。毕竟, 我们都是程序员, 源代码都在那里, 并且 shell 不过是一种普通的用户程序, 所以所有对它感兴趣的人都会去“修理”它。这非常疯狂, 后来我们不得不进行控制……”

1982年, 另一名贝尔实验室的科学家 David Korn 创建了一种 Bourne shell 的替代品, 称为 **Korn Shell** 或者 **ksh**。新 shell 基于 Korn 和其他研究人员在最近几年创建的工具。这样, 相对于标准 Bourne shell, 新 shell 进行了极大的改进。特别是, 新的 Korn shell 提供了历史文件、命令编辑、别名以及作业控制等特性(本书后面将讨论这些特性)。

Korn 确保 **ksh** 程序向后兼容当前的 **sh** 程序, 在很短的时间内, Korn shell 就成为贝尔实验室中的事实标准。在 Unix 的下一版发行时, Korn shell 分发到世界范围, 很快它就永久替代了 Bourne shell。从那时起, Korn shell 又发行了两个新的主要版本: 1988 年的 Ksh88 和 1993 年的 Ksh93。

在 20 世纪 90 年代初, 大量的压力要求标准化 Unix(参见第 2 章)。该压力导致了两个不同的运动, 其中一个由各种组织和委员会控制, 另一个起源于大众的需求。每个运动都有自己解决下述问题的方法: 如何一次性永久地标准化 Unix 的 shell?

“官方”运动创建了一大组规范, 即 POSIX(发音为“pause-ix”), POSIX 是操作系统标准化的蓝图。从实用来看, 可以认为 POSIX 是根据商业利益有组织地标准化 Unix 的一种尝试。

名称含义

POSIX

Unix 标准化项目最初由 IEEE 组织负责。起初, 该项目称为 IEEE-IX, 但是这是一个不好听的名称, 所以 IEEE 试图找一个好一些的名称。他们在这一问题上遇到了麻烦, 在最后一刻, 自由软件基金会(参见第 2 章)的创始人 Richard Stallman 建议称为 POSIX。Stallman 根据“Portable Operating System Interface”的首字母缩写起了这个名字。后面添加的“X”使这一名称看上去更像是 Unix 的名称。

POSIX 标准的一个重要部分就是 shell 基本特性的规范。多年以来, 这一标准曾经拥有过好几个名称, 包括 IEEE 1003.2 和 ISO/IEC 9945-2^{*}。对于公司、政府和其他组织来说, 1003.2 标准是一个重要的工作平台, 它为 shell 开发给出了一个明确定义的基准目标。例如, Ksh93 就是遵循 1003.2 标准设计的。

但是, 个人并不能直接获得 1003.2 标准。实际上, 1003.2 标准需要花钱才能获得一份技术细节副本^{**}。对于大多数 Unix 程序员来说, 占优势的理念并不是遵循一个诸如 POSIX 之类的统一标准, 而是创建其他人可以自由修改和增强的自由软件。正如第 2 章中所讨论的, 自由软件运动导致了自由软件基金会的发展和 Linux 的创建。但是 Korn shell 不能与 Linux 一起发行。问题在于虽说 Korn shell 是 Unix 的一部分, 但它是 AT&T 公司的商业产品。这样, 普通大众就不能使用它。

2000 年, AT&T 公司最终允许 Korn shell 成为一个开放源代码的产品, 但是为时已晚。在 20 世纪 90 年代, 已经创建了许多免费、开放源代码的 shell, 其中最重要的 shell 包括

^{*} 如果您希望了解一下这些缩写词, 下面分别介绍: IEEE 是 Institute of Electrical and Electronics Engineers(电气和电子工程师协会)的缩写。ISO 是 International Organization for Standardization(国际标准化组织)的缩写。名称 ISO 并不是一个只取首字母的缩写词。它取自希腊单词 isos, 含义为“相等”。最后, IEC 是 International Electrotechnical Commission(国际电工委员会)的缩写。

^{**} ISO 现在仍然采用这一方式。至于 IEEE, 则可以在线查看 1003.2 标准, 1003.2 标准是 IEEE 1003.1 规范的一部分。您可以到网站 www.unix.org 上查找“Single UNIX Specification”的链接, 所需的那一部分叫“Shell & Utilities”。

FreeBSD shell、Pdksh、Zsh 和 Bash。所有这些 shell 都遵循 1003.2 标准，从而使它们足以替换 Korn shell。

顾名思义，FreeBSD shell 是 FreeBSD 的默认 shell。为了保持传统，它也被命名为 **sh**——Bourne shell 家族成员的标准名称。换句话说，如果您使用的是 FreeBSD，那么在运行 **sh** 程序时，得到的 shell 是 FreeBSD shell。

Pdksh 是 Korn shell 的一个现代克隆体。Pdksh 编写的目的是为了提供一个没有许可限制的 Korn shell，因此才命名为“public domain Korn shell”。对 Pdksh 最好的定位就是认为它是一个现代的 Korn shell，既免费又开放源代码。最初的 Pdksh 由一位名叫 Eric Gisin 的程序员于 1987 年编写，基于由 Charles Forsyth* 编写的 Unix 公共领域第 7 版中的 Bourne shell。多年以来，许多人参与到 Pdksh 项目中去。但是，自 20 世纪 90 年代中期以后，Pdksh 就停止不前，几乎没有什么改变。这是因为(1)它工作得很好；(2)开放源代码社区中的大部分人们使用 Bash(参见下面)。然而，许多 Linux 系统依旧将 Pdksh 作为安装的 shell 之一，如果您的系统上有的话，可以试一试。

Bourne shell 家族的下一个重要成员是 Zsh，发音为“zee-shell”(在英国和加拿大，字母 Z 通常发音为“zed”)。Zsh 程序的名称是 **zsh**。

Zsh 是 Paul Falstad 在 1990 年开发的，那时他还是普林斯顿大学的一名本科生。他的原则是“从各个 shell 中吸取每样感兴趣的东西”。正如他自己的解释：“我希望它能做您希望做的任何一件事。”结果是 Zsh 提供了其他 Unix shell 的全部重要特性，以及还没有广泛应用的新能力。例如，Zsh 可以向您通报特定用户标识的登录。

那么 Zsh 这一名称是如何得来的呢？当 Falstad 开发这个 shell 时，有一个叫 Zhong Shao 的助教，他的 Unix 用户标识是 **zsh**。Falstad 觉得这是一个不错的名称，适合用作新创建 shell 的名称。

Zsh 发布不久，它就在世界范围内时兴起来，并在程序员和高级 Unix 用户之间广为流行。现在，Zsh 的现状与 Pdksh 极为相似：它有效、稳定，是一个出色的 shell，但是，自 20 世纪 90 年代中期开始，Zsh 的发展就变得非常缓慢，几乎停滞不前。

在所有 Bourne shell 家族的成员中，目前最重要的 shell 是 Bash。Bash 最初由 Brian Fox(1987 年)创建，后来(从 1990 年开始)由 Chet Ramey 维持，这些工作全部由自由软件基金会赞助。现在，Bash 受到遍布世界各地的程序员的支持。实际程序的名称，不用说也可以猜到，是 **bash**。

Bash 采用与 Korn shell 相似的方式扩展了基本的 Bourne shell 的功能。Bash 不仅是一个拥有强大脚本语言的命令处理器，而且还支持命令行编辑、命令历史、目录栈(directory stack)、命令自动补全、文件名自动补全以及许多其他特性(最终您会理解所有的特性)。

Bash 是一个自由软件，由自由软件基金会发行。它是 Linux 以及基于 Unix 的 Macintosh 的默认 shell，而且还可以在 Microsoft Windows 中使用(在一个类似于 Unix 系统的 Cygwin 下运行)。实际上，世界上每个重要的 Unix 系统或者提供 Bash，或者可以从 Internet 上免

*. 在 20 世纪 70 年代，我是加拿大 Waterloo 大学的一名本科生，学习数学和计算机科学。在一段时间内，我和 Charles Forsyth 共一间宿舍。他沉着冷静、容易相处、有点古怪，而且非常非常聪明。或许描述他最好的方式就是，他是 20 世纪 70 年代中期一名年轻的程序员，他看上去就像那种某一天会编写自己 shell 的那种人。

费下载一个相应的 Bash 版本。基于所有这些原因, Bash 是历史上最流行的 shell, 全世界有数百万人使用它。

名称含义

Bash

名称 Bash 代表 “Bourne-again shell”, 所以它是一个只取首字母的缩写词, 并且还是一个双关语。Bash 的思想就是——无论是从字面上还是内涵上讲——Bash 是一个基于标准 Unix shell 的复兴(born again, 也就是再生)版本。

注意, 尽管我们称以前的 shell 为 Korn shell 或者 C-Shell 或者 Zsh, 但是我们从来不说 Bash Shell, 而总是说 “Bash”。

11.3 C-Shell 家族: csh、tcsh

正如前面所述, 最初的 Bourne shell 于 1977 年开始使用。一年之后, 在 1978 年, 加利福尼亚大学伯克利分校的一名研究生 Bill Joy 开发了一种全新的 shell, 它基于 Unix 第 6 版的 **sh** 程序开发, 该程序是 Bourne shell 的前驱。

但是, Joy 并不是简单地复制已有的特性: 他添加了许多重要的改进, 包括别名、命令历史和作业控制。另外, 他还完全修补了编程工具, 改变了脚本语法的设计, 从而使其语法类似于 C 语言。基于这一原因, 他称他的新 shell 为 **C-Shell**, 而且将程序的名称由 **sh** 改为 **csh**。

20 世纪 70 年代末期, 以及整个 20 世纪 80 年代, C-Shell 非常流行。您可能会奇怪, 为什么在已经拥有其他优秀 shell 时还会出现这种情况呢? 原因有几个方面。

首先, C-Shell 对标准 Unix shell 进行了重大的改进。其次, C-Shell 作为组成部分包含在 BSD Unix 的发行版本中(参见第 2 章), 而 BSD Unix 本身就非常流行。最后, C-Shell 是由 Bill Joy 创建的, 而 Bill Joy 是最重要的 Unix 程序员之一。Joy 的工具, 包括 **vi** 编辑器, 设计都相当出色, 有许多人在使用(关于 Joy 对 Unix 贡献的讨论, 请参见第 2 章)。

很长一段时间, C-Shell 是大学和研究机构中 Unix 用户的 shell 选择, 通常是默认的 shell。我在 1976 年第一次使用 Unix, 而 C-Shell 就是我使用的第一个 shell。时至今日, 它仍然留在我的心里。

但是, C-Shell 还存在两个重要的问题。一个问题是可解决的; 然而另一个问题却无法解决, 从而导致 C-Shell 被有经验的用户所放弃。

首先, 因为 BSD 的许可条款, C-Shell 不能自由地发行和修改, 这对许多程序员来说是一个大问题。基于这一原因, 在 20 世纪 70 年代末, 一名来自卡内基梅隆大学的程序员 Ken Greer 开始着手编写一个完全自由的 **csh** 版本, 称之为 **tcsh**。在 20 世纪 80 年代初期, 开发 Tcsh(发音为 “Tee sea-shell”)的职责交给了由美国俄亥俄州的 Paul Placeway 领导的一个程序员小组。

Tcsh 令人惊奇。这不仅因为它是免费的(它在公共领域发行), 而且因为它增强了 C-Shell, 提供了许多高级特性, 例如文件名自动补全和命令行编辑等。Tcsh 吸引了大量的用户, 并且随着时间的流逝, 吸引了大批的志愿者维护和扩展它。

但是, C-Shell 还有一个无法解决的问题: C-Shell 和 Tcsh 不像 Bourne 家族的 shell 那样擅于编程。尽管类 C 的语法适于编写 C 程序,但是它并不适合于编写 shell 脚本,特别是涉及到 I/O(输入/输出)方面的内容时。此外, C-Shell 和 Tcsh 都有许多设计缺陷,这些缺陷太秘密了,以至于不能提及,否则会使那些对这些秘密担心的程序员异常愤怒。

到 20 世纪 90 年代,所有的流行 shell 都可以在所有的 Unix 系统上使用,这引起了一场关于哪个 shell 是最好的 shell 的广泛争论。就像年轻女士的衣服会由于自己的外貌而显得有点俗气一样, C-Shell 也不知由于什么原因开始失去它的名声。在 Unix 中坚用户以及他们的崇拜者中,开始流行这样的说法:“我喜欢在日常工作中使用 C-Shell,而在编程时使用 Bourne shell。”

1995 年,一名受人尊敬的 Unix 程序员,并且还是 Perl 编程语言之父之一的 Tom Christiansen 撰写了一篇广泛传播的论文,这篇论文的标题是“*Csh Programming Considered Harmful*”。该论文的标题来自一篇非常有名的计算机论文:“*Go To Statement Considered Harmful*”,一篇由荷兰程序员 Edsger Dijkstra 撰写的小论文^{*}。1968 年, Dijkstra 的论文改变了编程世界,致使结构化编程开始普及。1995 年, Christiansen 的论文,尽管没有如此重大的影响,但是却导致了人们在编程方面最终放弃 C-Shell 和 Tcsh。

现在, C-Shell 和 Tcsh 已经不像以前那样广泛使用了,自 20 世纪 90 年代中期开始, Tcsh 的开发缓慢得几乎接近于停止状态。然而, C-Shell 家族依然受许多高级 Unix 用户的高度关注。基于这一原因,本书中有一章专门介绍如何使用该 shell,但愿您愿意学习。

在一些系统上, **csh** 和 **tcsh** 是两个单独的程序。但是,在许多 Unix 系统上, **tcsh** 已经完全取代了 **csh**。也就是说,如果运行 **csh**,实际上得到的是 **tcsh**。通过查看 **csh** 的说明书页(参见第 9 章)可以辨别系统是不是这种情况。

即使是现在, C-Shell 仍然重要。实际上,当查阅其他 shell 时,总是能够看到取自 Bill Joy 多年以前在 C-Shell 中设计的特性。例如,在 Zsh 的网站上,上面写着:“Many of the useful features of **bash**, **ksh** and **tcsh** were incorporated into **zsh**(**bash**、**ksh** 和 **tcsh** 中的许多有用特性都已经集成到 **zsh** 中了)。”

名称含义

C-Shell、Tcsh

名称 C-Shell 来源于这一事实,即 Bill Joy 将该 shell 的编程工具设计成类似于 C 编程语言的工具。我猜测 Joy 喜欢“C-Shell”这一名称是因为它听起来像“sea shell”,有谁不喜欢海贝壳呢?

那么 Tcsh 是怎么来的呢?

当 Ken Greer 为 Unix 编写最初的 Tcsh 时,他也在使用一个叫 TENEX 的操作系统,在 DEC 公司的 PDP-10 计算机上运行。TENEX 命令解释器使用非常长的命令名称,因为这样

^{*} *Communications of the ACM (CACM)*, Vol. 11, No. 3, March 1968, pp 147-148。当您有空时,请阅读一下这篇论文(在 Internet 上很容易找到)。在阅读过程中,思考下述事实,即这篇短小、只有 14 段的论文比此前或者此后的任何一个法令对编程世界的改变都大。如果您还有兴趣,可以阅读 Dijkstra 一本独创性的书: *A Discipline of Programming*(Prentice-Hall PTR, 1976)。

顺便说一下, Dijkstra 的论文标题(以及后面 Christiansen 论文的标题)实际上是由 Niklaus Wirth 加上的,而不是 Dijkstra。Wirth 是 Algol W、Pascal 和 Modula-2 编程语言的创建者,那时他是 CACM 的编辑。

容易理解。但是，这些命令键入起来很麻烦，因此 TENEX 提供了一种功能，叫做“命令自动补全”，来完成大量的工作。您所需做的就是键入几个字母，然后按<Esc>键。然后命令解释器将把键入的内容补全成完整的命令。

Greer 将这一特性添加到新 C-Shell 中，当要命名该 shell 的名称时，他将该 shell 称为 **tcsh**，其中字母“t”指的就是 TENEX。

名称含义

C、C++、C#

C-Shell 和 Tcsh 都根据 C 编程语言命名。那么，为什么编程语言要起这样一个古怪的名称呢？

1963 年，一个叫 CPL 的编程语言在英国开发出来，该语言是剑桥大学和伦敦大学研究人员合作项目的一部分。CPL 代表“Combined Programming Language”，并且基于 Algol 60 语言，Algol 60 语言是第一个设计合理的现代编程语言。

4 年以后，在 1967 年，剑桥大学一名叫 Martin Richards 的程序员创建了 BCPL，即“Basic CPL”。BCPL 本身引起了另一个语言的产生，这个语言就是 B 语言。

B 语言传到贝尔实验室后，Ken Thompson 和 Dennis Ritchie 对它进行了修改，并重新命名为 NB。在 20 世纪 70 年代初期，在 Unix 的第 2 版中，Thompson(最早的 Unix 开发人员)使用 NB 重新编写了 Unix 的基本组。在那之前，所有的 Unix 都是用汇编语言编写的。不久之后，NB 语言就进行了扩展，并被重新命名为 C。C 语言很快就成为编写新 Unix 实用工具、应用程序，甚至是操作系统本身的语言选择。

人们可能会问，名称 C 是从哪里得来的呢？因为它是字母表中 B 之后的下一个字母，还是因为它是 BCPL 中的第二个字母呢？这一问题有着极其深刻的影响：C 语言的后继者是命名为 D 还是 P 呢？

最终的事实证明这一问题没有任何实际意义，在 20 世纪 80 年代初期，Bjarne Stroustrup(也是贝尔实验室的)设计了最流行的 C 语言扩展版本，一种面向对象的语言，它称之为 C++(发音为“C-plus-plus”)。在 C 语言中，++是一个运算符，即将变量加 1。例如，将变量 **total** 加 1，可以使用命令 **total++**。

2002 年，微软公司创建了一个特殊版本的 C++，作为其 .NET 倡议的一部分。他们称这个新语言为 C#，发音为“C sharp”(就像音乐术语一样)。如果您喜欢，可以把它想象成两个小的“++”标识，一个落在另一个上面，形成“#”符号。

因此，当看到名称 C、C++或者 C#时，您可以将它们看成奇妙的编程双关语，这些双关语使人们抓破脑袋也想不通人类到底是不是自然的主宰者。

11.4 应该使用哪种 shell

Unix 的 shell 不下数十种，我们已经讨论了最重要的 shell，包括 Bourne shell、Korn shell、FreeBSD shell、Pdksh、Zsh、Bash、C-Shell 和 Tcsh。

为了便于参考，图 11-1 示范了这些 shell 以及相应的程序名称*。当运行 shell 时，如果系统中存在该 shell，那么只需键入这个 shell 的名称即可，例如：

```
bash
ksh
tcsh
```

在一些 Unix 系统上，各种不同的 shell 都按照它们各自的名称安装。**sh** 程序不同于 **ksh** 或 **bash**，**cs**h 不同于 **tcsh**。因此，如果想使用一种老式的 Bourne shell，那么您要键入 **sh**；如果想使用 Bash，那么您要键入 **bash**。同理，您也可以使用 **cs**h(标准的 C-Shell)或者 **tcsh**(增强的 C-Shell)。

Shell	程序名称
Bash	bash 或者 sh
Bourne Shell	sh
C-Shell	cs h
FreeBSD Shell	sh
Korn Shell	ksh 或者 sh
Pdksh	ksh
Tcsh	tc sh 或者 cs h
Zsh	zsh

图 11-1 Unix shell

Unix shell 有许多种。本表列举了最常见的 shell，以及 shell 对应的实际程序名称。

但是，在一些系统上，比较新的 shell 已经替代了比较古老的 shell，因此无法再找到 Bourne shell 或者 C-Shell。此时，如果键入 **sh**，那么得到的 shell 或者是 Bash，或者是 Korn shell；如果键入 **cs**h，那么得到的 shell 是 Tcsh。要查看系统是不是这种情况，只需查阅需要了解的 shell 的说明书页。例如，在 Linux 系统上，如果请求 **sh** 的说明书页，得到的是 **bash** 的说明书页；如果请求 **cs**h 的说明书页，得到的是 **tcsh** 的说明书页。

显示特定 shell 的说明书页，可以使用 **man** 命令(参见第 9 章)，后面跟合适的程序名称。例如：

```
man sh
man cs
```

因为 shell 非常复杂，所以 shell 的说明书页实际上就像一个小的手册。不要被看到的内容所吓倒：其中大多数的内容是高级用户的参考资料。

* 另外还有两个常见的名称，您可能会遇到，它们看上去像普通的 shell，但实际上并不是，这两个名称是 **ssh** 和 **rsh**。**ssh** 程序是“Secure Shell(安全 shell)”，用于连接远程计算机。**rsh** 程序是“Remote Shell(远程 shell)”，它是一个比较古老的程序，现在已不推荐使用该程序，该程序用来在远程计算机上运行一条单独的命令。

那么, 应该使用哪一种 shell 呢? 如果您是一名初学者, 那么使用哪一种 shell 无关紧要, 因为所有的 shell 都拥有相同的基本特性。但是, 当您经验日益丰富时, 各种 shell 之间的细节就有关系了, 需要认真衡量一下。

如果您喜欢随大流, 坚持使用系统的默认 shell, 即键入 **sh** 时在您的系统上运行的 shell, 那么所使用的 shell 极有可能在 Linux 中是 Bash, 在 FreeBSD 中是 FreeBSD shell, 而在商业 Unix 系统中是 Korn shell。

但是, 如果您喜欢新奇, 那么有许多 shell 可以尝试。例如, 尽管 C-Shell 已经不再是默认的 shell, 但是还有许多人乐于使用它, 而且一般情况下, 系统中已经安装了 **tcsh** 和 **csh**。如果您喜欢使用那些不熟悉的 shell, 可以在 Internet 上搜索 Unix shell 或者 Linux shell, 我保证您会发现一些新的 shell(如果喜欢冒险, 但是无法决定使用哪一个, 可以试试 Zsh)。

提示

对于日常工作, 您可以根据自己的喜好选择任意一种 shell, 而且还可以自由地修改 shell。

但是, 如果编写 shell 脚本, 那么您应该坚持使用标准的 Bourne shell 编程语言, 以确保脚本能够移植到其他系统上。

一会儿之后, 我将示范如何修改 shell。但是在此之前, 我希望先回答一个有趣的问题: 每种 shell 有多复杂呢? 一个复杂的程序拥有大量的功能, 但是它也要求更多的时间来掌握。此外, 像许多 Unix 程序一样, shell 也拥有许多神秘的特性和选项, 这些特性和选项您可能永远不需要。这些额外的工具经常会分散人的注意力。

度量程序复杂性的一种粗略方法就是查看文档资料的长度。图 11-2 中的表列出了每种 shell 手册中所含字节(字符)的近似数量。为了便于比较, 我标准化了各个数字, 指定最小的数字为 1.0(当然, 当新版本的文档资料发布时, 这些数字可能会发生变化)。

shell 名称	说明书页大小	相对复杂度
Bourne Shell	38 000 字节	1.0
FreeBSD Shell	57 000 字节	1.5
C-Shell	64 000 字节	1.7
Korn Shell	121 000 字节	3.2
Tcsh	250 000 字节	6.6
Bash	302 000 字节	7.9
Zsh	789 000 字节	20.8

图 11-2 各种 shell 的相对复杂度

一种估计程序复杂度的方法就是查看文档资料的大小。本图中的统计数据显示了最流行的 Unix shell 的相对复杂度。

从这些数字很容易看出 C-Shell 和 FreeBSD Shell 的复杂度, 处于较古老的、功能较差的 Bourne shell 和其他较复杂的 shell 之间。

当然, 有人可能认为这无关紧要, 因为所有现代的 shell 都向后兼容, 或者兼容于 Bourne

shell, 或者兼容 C-Shell。如果不喜欢这些附加的特性, 可以不去理它们, 这也不会有什么麻烦。

但是, 文档资料相当重要。越复杂的 shell 的手册, 越需要时间去阅读, 而且也越难理解。实际上, 即便是 Bourne shell, 它的手册也很大, 常人难以细读。因此, 好好看一看图 11-2 中的数字, 仔细思考一下下面的问题:

假定 FreeBSD shell、Bash 和 Zsh 拥有 Unix 用户所需的大部分现代特性, 那么什么类型的人会选择使用 FreeBSD 和 FreeBSD shell 呢?

如果一个人选择使用 Bash 只是因为 Bash 是系统上默认的 shell, 那又怎样呢?

您认为什么类型的人喜欢下载、安装及学习一种像 Zsh 一样复杂的工具(这种工具并不是系统上默认安装的呢)?

11.5 临时改变 shell

当登录系统时, 系统会自动启动一个 shell。这个 shell 就是登录 shell。这里, 一个需要回答的重要问题是: 您的登录 shell 是哪一种 shell? 是 Bash、Korn shell、C-Shell 还是 Tcsh?

除非改变了默认 shell, 否则登录 shell 就是系统为您的用户标识所赋予的 shell。如果使用的是 Linux, 那么登录 shell 可能是 Bash。如果使用的是商业 Unix, 那么登录 shell 可能是 Korn shell。如果使用的是 FreeBSD, 那么登录 shell 可能是 Tcsh。

如果使用的是共享系统, 那么系统管理员会设置登录 shell。众所周知, 在自己的计算机上, 自己就是系统管理员(参见第 4 章), 因此, 除非自己改变了默认的 shell, 否则就是安装系统时自动设置的 shell。

改变 shell 的方式有两种。有时候, 您可能希望临时使用一个不同的 shell, 也许只是为了体验一下。有时候, 您可能发现一个比现在所用 shell 更好的新 shell, 所以希望永久地改变默认 shell。

例如, 假如您决定体验一下 Zsh, 于是从 Internet 上下载 Zsh 并安装在自己的系统上。刚开始, 您只是为了好玩, 临时将 shell 改变为 Zsh。最终, 您喜欢上了 Zsh, 因此决定将 Zsh 作为永久的登录 shell。在本节中, 将示范如何临时改变 shell。在下一节中, 将示范如何永久地改变 shell。

在开始之前, 记住 shell 就是一个程序, 可以像其他程序一样运行。这意味着, 在任何时候, 都可以停止当前的 shell, 并通过简单地运行一个新 shell 启动另一个 shell。例如, 假设您刚刚登录, 登录 shell 是 Bash。您输入了几条命令, 然后决定试试 Tcsh(假定您的系统上有该程序)。只需输入:

```
tcsh
```

当前 shell(Bash)暂停, 新 shell(Tcsh)启动。现在就可以使用 Tcsh 处理工作了。当准备切换回 Bash 时, 只需按 ^D(参见第 7 章)表示不再有数据了即可。这时 Tcsh shell 结束, 然后返回到最初的 Bash shell。整个过程就是这样简单。

如果想体验一下，可以先使用下述命令查看系统上安装了哪些 shell：

```
less /etc/shells
```

我们将在下一节中讨论这个命令。

如果您正在体验，但不知道正在使用的是哪个 shell，则可以在任何时间使用下述命令显示当前 shell 的名称：

```
echo $SHELL
```

提示

启动一个新 shell 后，接着再启动一个新 shell，然后再启动一个新 shell，如此重复是可能的。但是，当要结束工作会话时，只能从原始登录 shell 注销。

因此，如果启动了不止一个新 shell，则必须原路返回到登录 shell 才能注销。

11.6 口令文件，改变登录 shell: chsh

Unix 有两个文件用来描述系统中所有的用户标识。第一个文件是 `/etc/passwd`，即口令文件，包含每个用户标识的基本信息。第二个文件是 `/etc/shadow`，即影子文件，包含每个用户标识的实际口令(当然是加密的)。

当登录时，Unix 从这两个文件中检索与您的用户标识相关的信息。特别是 `/etc/passwd` 文件，该文件包含有登录 shell 的名称。因此，为了修改登录 shell，所需做的工作就是对 `/etc/passwd` 文件进行一个简单的修改。但是，该修改不能直接进行。这样做太危险了，因为破坏 `/etc/passwd` 文件会严重破坏系统。实际上，修改该文件需要使用一个特殊的命令，稍后再解释这个命令。

在此之前，我希望先介绍一些关于文件名称的信息。前面书写刚提及的两个文件的方式称为“路径名”。路径名说明一个文件在文件系统中的准确位置。当在第 23 章中讨论 Unix 文件系统时，我们再详细讨论这一思想。现在，您需要知道的就是路径名 `/etc/passwd` 指 `/etc` 目录中的 `passwd` 文件。目录就是 Windows 或者 Mac 用户所谓的文件夹(这一思想以及其他许多思想都取自 Unix)。

当改变 shell 时，需要以路径名指定 shell 程序的名称。可用 shell 的路径名存储在一个名为 `/etc/shells`^{*} 的文件中。要显示该文件，可以使用 `less` 命令(我们将在第 21 章中正式讨论这个命令)。

```
less /etc/shells
```

下面是一个典型输出：

```
/bin/sh
/bin/bash
/bin/tcsh/bin/csh
```

^{*} Linux 和 FreeBSD 使用 `/etc/shells` 文件，但是某些商业 Unix 系统(例如 AIX 和 Solaris)并不使用这个文件。

从该输出中可以看出, 在这个例子中有 4 个可用的 shell: **sh**、**bash**、**tcsh** 和 **csh**。改变登录 shell 时应使用 **chsh**(change shell, 改变 shell)命令。该命令的语法*为:

```
chsh [-s shell] [userid]
```

其中 *userid* 是希望改变其 shell 的用户的用户标识, *shell* 是新登录 shell 的路径名。

注意: (1)只能改变自己的用户标识的 shell。如果要改变另一个用户标识的 shell, 则必须是超级用户。(2)在一些系统上, 除非希望使用的 shell 列举在/etc/shells 文件中, 否则不允许 **chsh** 改变到该 shell。而在其他系统上, **chsh** 允许进行这样的修改, 但是会提示一个警告。因此, 如果在自己的系统上下载并安装了一个 shell, 那么最好将该 shell 的路径名添加到/etc/shells 文件中。

默认情况下, **chsh** 假定您希望改变自己的登录 shell, 因此不用指定用户标识。例如, 将自己的登录 shell 修改为/bin/tcsh, 可以使用命令:

```
chsh -s /bin/tcsh
```

如果没有输入-s 选项和 shell 的名称, 那么 **chsh** 将提示(也就是要求)输入-s 选项和 shell 的名称。例如, 假设您作为 **harley** 登录, 而且输入了:

```
chsh
```

那么您将看到:

```
Changing shell for harley.  
New shell [/bin/bash]:
```

此时 **chsh** 告诉您两件事情。首先, 您将会修改用户标识 **harley** 的登录 shell。其次, 当前的登录 shell 是/bin/bash。

现在您有两个选择。如果输入新 shell 的路径名, **chsh** 将为您改变登录 shell。例如, 将当前登录 shell 修改为 Tcsh, 可以输入:

```
/bin/tcsh
```

如果您只是按了<Enter>键, 而没有键入任何内容, 那么 **chsh** 不会做任何改变。这是一种查看当前登录 shell 的好方法。

至于 Linux, **chsh** 命令还有另外一个有用的选项: 使用-l(list, 列举)选项可以显示当前可用的全部 shell:

```
chsh -l
```

一些 Unix 系统没有 **chsh** 命令。此时, 要使用 **passwd** 命令(在第 4 章中讨论过)的一个变体来修改登录 shell。例如, 对于 AIX 来说, 使用 **passwd -s**; 对于 Solaris 来说, 使用 **passwd**

* 在本书中, 我讲授的内容包括大量命令的使用方法。每描述一条新命令, 我将首先说明该命令的语法。在说明命令的语法时, 只说明那些最重要的选项和参数。如果您需要更多的信息, 可以查阅命令的说明书页(参见第 9 章)。

正如第 10 章中解释的, 一些命令在各个版本的 Unix 之间存在着很小的差别。在本书中, 我通常使用 GNU/Linux 版本的命令。大多数时候, 这没有什么问题, 因为重要的选项和参数在大多数类型的 Unix 中都是相同的。但是, 如果您遇到了问题, 那么我希望您记住, 您的系统上关于一个程序如何运转的最终参考资料就是联机手册。

-e。更多的信息，请参见 **passwd** 命令的说明书页。

如果您是一名系统管理员，有人请求您改变他人的登录 shell，可以使用 **usermod -s**。该命令的更多信息，请参见说明书页。

提示

当改变登录 shell 时，修改的是 **/etc/passwd** 文件。因此，无论怎么修改，都要等到下一次登录系统时才能生效(就像改变口令一样)。

提示

在一些特定情形中，如果系统处于危急状态，而您需要以超级用户(**root**)登录进行修复，那么有一些 shell 可能不能正常运行。基于这一原因，**root** 的登录 shell 必须是总能够正常运行的那一种 shell，而不管它是什么类型的。

因此，除非您真的知道自己在做什么，否则不要修改用户标识 **root** 的登录 shell。如果这样的话，您可能会处于这样的危险中——有一天，您的系统损坏了，而您却没有办法登录系统。

11.7 练习

1. 复习题

1. 什么是 shell?
2. shell 的两大家族各指什么? 列举每个家族中最常使用的 shell。
3. “向后兼容”的含义是什么? 举一个 shell 向后兼容另一个 shell 的例子。
4. 什么是 POSIX? 它如何发音?
5. 如何临时改变 shell? 如何永久地改变 shell?

2. 应用题

1. 查看您的系统上的可用 shell，显示默认 shell 的名称。
2. 将默认 shell 改变为另一种 shell。注销系统并再次登录。检查默认 shell 是不是已经成功改变。将默认 shell 改变回原来的 shell。注销系统并再次登录。检查默认 shell 是不是正确变回原来的 shell。

3. 思考题

1. 多年以来，人们创建了许多不同的 Unix shell。为什么这是必然的? 为什么不创建一个主 shell，然后不时地对它进行增强呢?
2. 程序 A 已经被程序 B 替代。如果程序 B 向后兼容程序 A，这种情况有什么优点? 又有什么缺点?

使用 shell：变量和选项

许多人不愿意花时间学习如何熟练地使用 shell，这是错误的。可以确定的是，shell 和其他复杂的 Unix 程序相似，拥有许多并不真的需要理解的特性。但是，shell 中有许多基本的思想拥有极大的实用价值。下面是具体列表：

- 交互式 shell
- 进程
- 环境变量
- shell 变量
- shell 选项
- 元字符
- 引用
- 外部命令
- 内置命令
- 搜索路径
- 命令替换
- 历史列表
- 自动补全
- 命令行编辑
- 别名
- 初始化文件
- 注释

如果您在查看这个列表时觉得很受打击，我可以理解。或许您想问：“我真的需要学习所有内容吗？”

答案是肯定的，而且还需要不短的时间，不过您不用担心。首先，我将把这些内容分散到 3 章中进行讨论，不会让您一下子学习很多的内容。其次，我将确保一个接一个地讨论这些主题，不会让您觉得混乱(实际上，我将按照上述列表的顺序讨论各个主题)。最后，随着您开始欣赏 shell 的魅力，并开始觉得所有的东西都很有意义，您将发现自己乐于学习各个主题。

众所周知，shell 共有两大家族：Bourne 家族(Bash、Korn shell)和 C-Shell 家族(C-Shell、Tcsh)。当学习如何使用 shell 时，根据所使用 shell 的不同，一些具体细节可能有所不同，

这些不同会反映在本书的讲述过程中。无论如何, 熟练掌握 Unix 的一个基本标准就是理解和欣赏每个 shell 家族如何解决特定的问题。

因此, 在学习接下来的 3 章内容时, 无论您使用的是哪一种 shell, 我希望您阅读所有各节的内容以及所有的示例。一些人只学习他们当时使用的 shell, 但这是不正确的。我的目标是让您熟悉所有主要的 shell。实现这一目标的方法就是关注基本原理, 而不是去记忆某个特定 shell 的深奥细节。

在阅读本章时, 您需要知道自己正在使用的是哪种 shell。如果您还不确定, 可以在登录系统时使用下述命令显示所使用 shell 的名称(本章后面将介绍它的含义):

```
echo $SHELL
```

如果您临时改变到另一种 shell(参见第 11 章), 那么您当然知道自己使用的是哪种 shell。

在开始之前明确最后一点: 第 11 章已经大体上介绍了一些 shell 的相关知识, 如果您还没有阅读第 11 章的内容, 那么请在继续阅读本章内容之前, 花一点时间阅读一下第 11 章的内容。

12.1 交互式 shell 和非交互式 shell

交互式程序指的是与人进行沟通的程序。当运行交互式程序时, 程序的输入来源于键盘或者鼠标, 而程序的输出发送到显示器。例如, 当使用字处理程序或者 Web 浏览器时, 所使用的就是交互式程序。

非交互式程序指的是独立于人运行的程序。一般情况下, 它从文件中获取输入, 并将输出写入到另一个文件中。例如, 当编译程序(对程序进行处理, 从而使程序可以运行)时, 使用的就是一个非交互式程序, 即编译器。

有时候, 交互式程序和非交互式程序之间的界限可能有点模糊。例如, 交互式程序可能将结果发送给文件或者打印机。同理, 非交互式程序也可能要求您使用键盘输入一些数据, 或者在发生重要事情时在显示器上显示一个消息。

然而, 实际上, 程序仍然可以简单地分为交互式程序(与人一起工作)和非交互式程序(独自工作)。通常, 交互式程序从人(通过键盘、鼠标)获取输入, 并且将输出发送给人(通过显示器、扬声器)。非交互式程序使用非人类资源(例如文件), 并且将输出发送给非人类资源(例如另一个文件)。因此, 这里出现了一个问题, shell 属于哪一种: 交互式还是非交互式的?

答案是两种都是。第 11 章中讲过, shell 既可以充当用户界面, 也可以作为脚本解释器。要使用 CLI(命令行界面), 需要打开一个终端窗口或者使用一个虚拟终端(参见第 6 章)。当见到 shell 提示时, 就可以输入命令。然后 shell 处理命令, 处理完命令后, shell 显示另一个提示。当以这种方式工作时, shell 就是用户界面, 因此我们可以说您使用的 shell 是一个交互式 shell。

另外, 您还可以创建一组命令(称为 shell 脚本), 并保存在一个文件中。当运行脚本时, shell 从文件中读取命令, 并且在不需要输入的情况下一并处理所有的命令。当发生这种情况时, 我们就称您使用的 shell 是一个非交互式 shell。

在这两种情况下，所使用的 shell 是同一类型的 shell，理解这一点非常重要。之所以能这样是因为 shell 被设计成既可以交互式工作，又可以非交互式工作。

登录时，系统会为您启动一个 shell，并设置成交互式的。同样，当从命令行手动启动一个新 shell 时(例如，通过键入 **bash** 或者 **tcsh**)，新 shell 也会被设置成交互式的。

另一方面，当运行 shell 脚本时，会自动启动一个新的 shell，并将解释脚本的任务交给该 shell 处理。这个新 shell 被设置为非交互式的。一旦任务完成，也就是说脚本执行完成，非交互式 shell 也就终止。

那么 shell 是如何知道它应该是交互式还是非交互式的呢？这取决 shell 启动时给它指定的选项。我们将在本章的后面讨论 shell 选项。

名称含义

shell

在日常生活中，我们以两种不同的方式使用单词“shell”，这两种方式可能有点混乱。我们可能讨论一般意义上的 shell，也有可能指一个正在运行的 shell 实例。

例如，您是一名年轻人，受邀出席一个有许多啦啦队队长参加的女学生联谊会。有人把您介绍给该联谊会上最漂亮的女孩，为了打破沉默，您问她：“您使用什么 shell？”交谈了几分钟之后，她说到：“足球运动员好烦人。我喜欢理解 shell 的人。您能不能到我的宿舍帮我优化一下内核？”在这个例子中，您和这个女孩所谈论的 shell 就是一般意义上的 shell。

第二天，您坐在教室里听教授讲 Unix 的课程，教授说：“.....在登录之后，系统就会启动一个 shell 充当用户界面。如果键入 **bash** 命令，就会启动一个新的 shell。当运行 shell 脚本时，又会启动另一个 shell.....”在这种情况下，教授不是指一般意义上的 shell。他在谈论正在运行的实际 shell。

为了确保您已经理解这一区别，下面请看看能不能理解这一句话：“一旦您学会了如何使用 shell，就可以在任何时候启动一个新 shell。”

12.2 环境、进程和变量

在第 6 章讨论多任务处理的过程中，我介绍了这样一种思想，即在 Unix 系统中，每个对象都被表示为一个文件或者进程。简单地讲，文件存放数据或者用来访问资源，而进程是正在执行的程序。因此，一个正在运行的 shell 就是一个进程。同理，任何从 shell 中启动的程序也都是一个进程。

在进程运行过程中，它需要访问所谓的环境，即一组用来存放信息的变量。为了理解这一思想，我们先讨论一个基本的问题：什么是变量，我们可以对变量做些什么？

首先讨论变量的定义。变量是一个用来存储数据的实体。每个变量都有一个名称和一个值。其中变量名是用来引用变量的标识符，值是存储在变量中的数据。

下面举例说明。正如第 7 章中讨论的，Unix 用户使用一个叫 **TERM** 的变量来存储所使用终端的类型名称。这样，任何需要知道终端类型的程序，只需简单地查看 **TERM** 的值即可。最常见的 **TERM** 值有 **xterm**、**linux**、**vt100** 和 **ansi**。

当命名变量时，您拥有很大的灵活性：只有两个简单的规则。首先，变量名必须由大

写字母(A-Z)、小写字母(a-z)、数字(0-9)或者下划线字符(_)构成。其次，变量名的第一个字符必须是字母或者下划线，不能是数字。因此，变量名 **TERM**、**path** 和 **TIME_ZONE** 都有有效的，而变量名 **2HARLEY** 是无效的。

当使用 Unix shell 时，有两种不同类型的变量。它们分别称为“shell 变量”和“环境变量”，我们将在本章中讨论这两种变量。通常，变量只有 4 种不同类型的操作，即创建变量、查看变量的值、修改变量的值以及销毁变量。

通俗地讲，可以将变量看作一个小盒子。变量名位于盒子上，而变量的值位于盒子里面。例如，您可以这样想象，即一个名为 **TERM** 的盒子，里面包含一个单词 **xterm**。在这个例子中，我们说变量 **TERM** 的值为 **xterm**。

对于大多数编程语言来说，变量可以包含许多不同类型的数据，有字符、字符串、整型数、浮点型数、数组、集合等。至于 shell，变量几乎总是存储一种类型的数据，即字符串，也就是一串纯文本字符。例如，在我们的例子中，变量 **TERM** 存储的字符串包含 5 个字符：**x**、**t**、**e**、**r** 和 **m**。

在创建变量时，尽管不是必须的，但还是通常会给它赋一个值。如果变量没有赋值，那么我们可以说这个变量拥有一个 **null** 值，这意味着这个变量没有值。这就好比创建了一个盒子，给这个盒子起了一个名称，但是没有在盒子里面放任何东西。如果变量拥有 **null** 值，那么在需要时，随时可以给这个变量赋一个值。

现在让我们看看变量、环境和进程是如何结合在一起的。考虑下述情形。您在 shell 提示处启动了 **vi** 文本编辑器(参见第 22 章)。用技术术语讲，我们可以说一个进程(shell)启动了另一个进程(**vi**)。(我们将在第 26 章中讨论有关进程的细节问题。)

当发生这种情况时，第一个进程称为父进程或者双亲，第二个进程称为子进程或者孩子。在这个例子中，父进程就是 shell，而子进程就是 **vi**。

在子进程创建时，系统为子进程复制了父进程的环境。我们说子进程继承了父进程的环境。这意味着父进程可以访问的全部环境变量，现在子进程也可以访问。

例如，在我们的例子中，当创建 **vi**(子进程)时，它继承了 **shell**(父进程)的环境。这种情况下，为了查明正在使用的终端类型，**vi** 现在能够查看 **TERM** 变量的值。这就使得 **vi** 可以正确格式化它的输出，以适应特定的终端。

12.3 环境变量和 shell 变量

如果您是一名程序员，那么您应该理解全局变量和局部变量之间的区别。在程序中，局部变量只存在于创建它的范围之内。例如，假设您在编写一个程序，创建了一个只在函数 **calculate** 中使用的变量 **count**，我们就可以说变量 **count** 是一个局部变量。更具体地讲，可以说变量 **count** 是函数 **calculate** 的局部变量。这意味着，当函数 **calculate** 运行时，变量 **count** 才存在；一旦 **calculate** 停止运行，变量 **count** 就不存在了*。

* 在本章中，我们只讨论存储一个值的简单变量。如果您准备编写 shell 脚本，那么您应该知道 Bash 和 Korn shell 都允许使用一维数组(数组是一种包含一系列值的变量)。更多的信息，请参见 Bash 的说明书页(查看“Arrays”一节)或者 Korn shell 的说明书页(查看“Parameters”一节)。

另一方面，全局变量可以在程序的任何地方使用。例如，假设您正在编写一个对一长串数字进行统计运算的程序。如果将这个数字列表定义为全局变量，那么它就可以在程序的所有地方使用。这意味着，如果一个函数或者过程对这个列表进行了修改，那么程序的其他部分也会看到该修改。

这里有一个问题：当使用 Unix shell 时，shell 中的全局变量和局部变量与程序员使用的全局变量和局部变量相似吗？

答案是肯定的。所有的 shell 都使用全局变量和局部变量，您需要知道它们的运转方式。首先，shell 中有环境变量，这些我们已经讨论过。因为环境变量对所有进程可用，所以它们是全局变量，实际上，我们通常称环境变量为全局变量*。

其次，有的 shell 变量只在特定的 shell 中使用，并不属于环境。因此，这些变量不从父进程传递给子进程，基于这一原因，我们称这种变量为局部变量。

通常，局部(shell)变量以两种方式使用。第一种方式，它们可能用来存放对 shell 本身有意义的信息。例如，在 C-Shell 和 Tcsh 中，shell 变量 `ignoreeof` 用来确定 shell 在用户按下 `^D`(参见第 7 章)时是否忽略 `eof` 信号。

第二种方式，shell 变量在 shell 脚本中以普通程序中局部变量的方式使用：作为临时存储容器。因此，在编写 shell 脚本的过程中，当需要临时存储时就可以创建这样的 shell 变量。

到目前为止，所有这些结论都比较直接。shell 变量是创建它们的 shell 的局部变量。环境变量是全局变量，因为使用相同环境的任何进程都可以访问它们。

但是，在实际中还有一个问题。这是因为，在 shell 中，局部变量和全局变量之间的界限是模糊的。基于这一原因，我希望花一些时间准确解释 shell 如何处理变量。此外，Bourne 和 C-Shell 家族之间有明显的区别，因此我们不得不分开讨论它们。但是，这些概念非常重要，因此我希望您理解两大 shell 家族中变量的工作方式，不管您现在使用的是哪一种 shell。

在开始之前，先解释一下变量是如何命名的。在一些编程语言中，有一个传统，即全局变量通常采用大写字母的名称，而局部变量采用小写字母的名称。C-Shell 家族(C-Shell、Tcsh)采用这种传统。环境变量采用大写字母的名称，如 `HARLEY`；shell 变量采用小写字母的名称，如 `harley`。

Bourne shell 家族(Bash、Korn shell)有所不同：shell 变量和环境变量传统上都采用大写字母的名称。为什么是这种情况呢？原因稍后再解释。

对于大多数编程语言来说，变量或者是局部的，或者是全局的。而对于 shell 来说，有一个奇怪的问题：一些变量同时拥有局部变量和全局变量的含义。换句话说，就是有一些变量对 shell 本身是有用的(这意味着这些变量应该是 shell 变量)，并且对由 shell 启动的进程也是有用的(这意味着这些变量应该是环境变量)。

Bourne shell 家族通过将变量或者只定义为局部变量，或者同时定义为局部和全局变量来解决这个问题。Bourne shell 家族的 shell 中没有完全的全局变量。例如，可能有两个变量 `A` 和 `B`，变量 `A` 是一个 shell 变量，变量 `B` 同时是 shell 变量和环境变量。但是，不能有

* 严格从编程意义上讲，环境变量并不完全是全局的，因为子进程对环境变量的修改不会传递到父进程中。

这一限制有一个合理的原因：允许子进程修改父进程的环境变量将产生极大的混乱、bug 和安全漏洞。

一个只属于环境变量的变量(仔细地思考这一点)。

那么在创建变量时又会发生什么情况呢?在 Bourne shell 家族中,只允许创建局部变量。也就是说,每个新变量被自动地设置为 shell 变量。如果希望某个变量同时成为环境变量,必须使用一个称为 **export** 的特殊命令。**export** 命令将 shell 变量修改为“shell+环境”变量。当这样做时,就可以称将变量导出(**export**)到环境中。

下面举例说明(现在还不必担心细节问题,本章后面会详细讨论)。开始时,我们创建一个命名为 **HARLEY** 的变量,并将值 **cool** 赋给它:

```
HARLEY=cool
```

此时, **HARLEY** 只是一个 shell 变量。如果启动一个新 shell 或者运行一条命令,则新进程并不能访问 **HARLEY**,因为它还不是环境变量。下面将 **HARLEY** 导出到环境中:

```
export HARLEY
```

现在 **HARLEY** 变量既是 shell 变量又是环境变量了。如果这时再启动一个新 shell 或者运行一条命令,那么它们就可以访问 **HARLEY**。

您现在应该明白为什么 Bourne shell 为 shell 变量和环境变量都使用大写字母了。使用大写字母可以使变量名称突出,而且因为没有纯粹的环境变量,所以没有简单的方法来区分局部变量和全局变量(再花一点时间仔细考虑一下)。

可以看出, Bourne shell 家族处理变量的方式有点使人感到混乱,特别是初学者。实际上,这些 shell 所使用的系统可以追溯到第一个 Bourne shell,由 Steve Bourne 于 1976 年在贝尔实验室开发(参见第 11 章)。两年之后,即 1978 年,当 Bill Joy 在加利福尼亚大学伯克利分校开发 C-Shell(参见第 11 章)时,他决定改进变量的组织方式。他创建了一个非常简明的系统,在这个系统中他对环境变量和 shell 变量进行了明显的区分。

在 C-Shell 家族中,环境变量通过命令 **setenv**(稍后描述)创建,并且以大写字母命名,如 **TERM**。shell 变量由 **set** 命令(稍后描述)创建,并且以小写字母命名,如 **user***。实际上,也就是这么一回事。

但是,C-Shell 系统的简单性留下了一个恼人的问题。正如前面所述,有一些变量在 shell 中以及所有的子进程中都有意义。Bourne shell 家族通过使用既是局部又是全局的变量避免了这一问题。但是,C-Shell 家族不允许这样做。

不过,C-Shell 家族也认识到一些特殊的变量需要既是局部变量又是全局变量。解决方法就是定义少数几个特殊的 shell 变量,这些变量分别绑定到对应的环境变量。无论何时,当这些变量改变时,shell 会自动地更新对应的变量。

例如,有一个名为 **home** 的 shell 变量对应于环境变量 **HOME**。如果修改 **home**,那么

* 很长一段时间,人们倾向于贬低 C-Shell,特别是在与现代的 Bourne shell(例如 Bash)对比时。在第 11 章中,当我们讨论 Tom Christiansen 的论文 *Csh Programming Considered Harmful* 时,已经讨论过这一文化信仰。

但是, Bourne shell 为了保持向后兼容,继承了许多严重的设计缺陷,而且还无法修改。例如,考虑 Bourne shell 处理局部变量和全局变量的混乱方式,C-Shell 尽管自身拥有缺陷,但是在这一点上反映了 Bill Joy 的先见之明。Bill Joy 是一名才华横溢的程序员,在年轻时拥有设计高质量工具的惊人天赋。

当选择自己的 shell 时,不要让他人过度地影响您。现代版本的 C-Shell(Tcsh)是一个出色的工具,对于交互式应用,它可以与 Bash 和 Korn shell 相媲美(或许有人会写一篇新论文,题目就叫 *Don't Bash the C-Shell*)。

shell 将对 **HOME** 进行同样的修改。如果修改 **HOME**，那么 shell 也将修改 **home**。

在所有拥有双名称的变量中，只有 5 个在日常应用中比较重要(参见图 12-1)。现在您应该已经理解了 **TERM** 和 **USER** 变量。**PATH** 变量将在本章后面解释。对于 **PWD** 和 **HOME** 变量，在讨论完 Unix 文件系统(第 23 章)和目录(第 24 章)之后，您就可以理解它们。

shell 变量	环境变量	含义
cwd	PWD	当前/工作目录
home	HOME	home 目录
path	PATH	搜索程序的目录
term	TERM	正在使用的终端类型
user	USER	当前用户标识

图 12-1 C-Shell 家族：shell/环境互通变量

在 C-Shell 家族中，认为有少数几个 shell 变量与对应的环境变量相同。当这一对变量中有一个变量改变时，shell 会自动地修改另一个变量。例如，当 **home** 变量改变时，shell 会自动地修改 **HOME** 变量，反之亦然。详情请参见正文。

名称含义
<p>cwd、PWD</p> <p>图 12-1 中示范了 C-Shell 中互通的变量对。可以看出，除一个特例之外，每个 shell 变量与对应的环境变量拥有相同的名称(不考虑字母的大小写)。这个唯一的特例就是 cwd 和 PWD。这两个变量包含工作目录的名称，有时候也称为当前目录(参见第 24 章)。因此，名称 cwd 就是：current/working directory(当前/工作目录)。</p> <p>PWD 变量根据 pwd 命令命名，该命令显示工作目录的名称。更有趣的是，pwd 是最初的 Unix 命令之一。它的含义是“print working directory(显示工作目录)”，而且它可以追溯到计算机输出真正打印在纸张上的时代(参见第 3 章和第 7 章)。</p>

12.4 显示环境变量：env、printenv

尽管您可以创建自己的环境变量和 shell 变量，但是除非编写程序，否则并不需要这样做。大多数时候，使用的是默认变量。

默认变量有哪些呢？为了显示默认变量，可以使用命令 **env**：

env

在许多系统上，还有另外一个命令可以使用，这个命令是 **printenv**：

printenv

当使用 **env** 或 **printenv** 命令时，环境变量可能有许多，以至于有些变量滚动出屏幕。如果出现这种情况，可以使用 **less** 命令每次一屏地显示输出：

env | less

```
printenv | less
```

当显示环境变量时，环境变量并不会根据字母表顺序排序。如果希望对输出进行排序，可以使用 **sort** 命令(参见第 19 章)，如下所示：

```
env | sort | less
printenv | sort | less
```

这种结构称为“管道线(pipeline)”。我们将在第 15 章和第 16 章中讨论管道线。

出于参考目的，图 12-2 显示了最重要的环境变量及其含义。在计算机上看到的实际变量可能会由于操作系统和所使用 shell 的不同而有所不同。但是，大多数变量会和表中的相同。如果不能理解所有的内容，不用担心，当您学习了关于环境变量使用的足够知识之后，就会理解它们的目的。

shell	变量	含义
B K . .	CDPATH	cd 命令搜索的目录
B K . T	COLUMNS	屏幕或者窗口的宽度(以字符为单位)
B K C T	EDITOR	默认文本编辑器
B K . .	ENV	环境文件的名称
B K . .	FCEDIT	历史列表: fc 命令使用的编辑器
B K . .	HISTFILE	历史列表: 用来存储历史命令的文件名称
B K . .	HISTSIZE	历史列表: 存储历史命令的最大数量
B K C T	HOME	home 目录
. . . T	HOST	计算机名称
B . . .	HOSTNAME	计算机名称
B . . T	HOSTTYPE	主机计算机的类型
B . . .	IGNOREEOF	在结束 shell 之前忽略的 eof 信号(^D)的数量
B K C T	LOGNAME	当前用户标识
B . . T	MACHTYPE	系统描述
B K C T	MAIL	查看新邮件的文件
B K C T	MAILCHECK	shell 查看新邮件的时间间隔(单位为秒)
B K . .	MAILPAT	查看新邮件的文件
B K . .	OLDPWD	前一个工作目录
B . . T	OSTYPE	操作系统的描述
B K C T	PAGER	显示数据的默认程序(应该是 less)
B K C T	PATH	搜索程序的目录
B K . .	PS1	shell 提示(通过修改这个变量进行 shell 提示的定制)
B K . .	PS2	连续行的特殊 shell 提示
B K C T	PWD	工作[当前]目录

图 12-2 最重要的环境变量

B K . .	RANDOM	0 至 32 767 之间的随机数
B K . .	SECONDS	自 shell 激活之后过去的时间(单位为秒)
B K C T	SHELL	登录 shell 的路径名
B K C T	TERM	正在使用的终端类型
B K . .	TMOUT	在不键入命令时,系统自动注销前的等待时间(单位为秒)
. K C T	TZ	时区信息
B K C T	USER	当前用户标识
B K C T	VISUAL	默认文本编辑器(覆盖 EDITOR)

图 12-2 (续)

默认情况下, Unix 系统使用大量的环境变量。系统中的环境变量依赖于所安装的操作系统和所使用的 shell。

最左边的一列说明哪个 shell 支持这个变量: B=Bash; K=Korn Shell; C=C-Shell; T=Tcsh。圆点表示相应的 shell 不支持该选项。

12.5 显示 shell 变量: set

使用不带选项或者参数的 **set** 命令可以显示所有的 shell 变量以及它们的值:

set

这个命令非常简单, 而且适用于所有的 shell。但是, 这里有非常重要的一点需要记住。

对于 C-Shell 家族来说, shell 变量都是小写字母的名称。按照定义, 它们都是局部变量。

对于 Bourne shell 家族来说, shell 变量都是大写字母的名称。因此, 只看变量的名称无法确定这个变量是局部变量还是全局变量。如果变量只是 shell 变量, 那么它就是局部变量; 如果变量既是 shell 变量又是环境变量, 那么它既是局部变量又是全局变量(记住, 在 Bourne shell 家族中, 没有纯粹的全局变量)。这意味着当使用 **set** 命令显示变量时, 没有什么简单的方法知道哪些变量已经导出到环境中。

提示

奇怪但真实的是: 确定哪些 Bourne shell 变量没有导出的唯一方法就是比较 **set** 命令和 **env** 命令的输出。如果变量位于 **set** 的输出中, 但是没有位于 **env** 的输出中, 那么这个变量就是 shell 变量。如果变量既位于 **set** 的输出中, 又位于 **env** 的输出中, 那么这个变量既是 shell 变量又是环境变量。

很明显, 这容易使人迷惑。但是, 这无关紧要, 因为在 Bourne shell 家族中并不怎么使用 shell 变量。诚然, 在编写 shell 脚本时, 有需要时就会创建局部(shell)变量。但是对于日常的交互式工作来说, 重要的是环境变量, 而不是 shell 变量。

在 C-Shell 家族中, 情况就不一样了。C-Shell 家族中有大量的 shell 变量, 其中有许多用来控制 shell 的行为。前面曾提到几个这种变量: **cwd**、**home**、**term** 和 **user**。出于参考

目的，图 12-3 列出了这 4 个变量，以及一些我认为最重要的变量。有关 C-Shell 家族中 shell 变量的完整列表，请参见附录 G(实际上，您现在可能就希望快速浏览一遍附录 G 的内容，看看 C-Shell 家族实际使用了多少个 shell 变量)。

shell	shell 变量	含义
• T	autologout	如果不键入命令，系统自动注销前的等待时间(单位为秒)
C T	cdpath	cd 、 chdir 、 popd 搜索的目录
• T	color	使 ls -F 命令使用颜色
C T	cwd	工作[当前]目录(与 owd 相对)
C T	filec	自动补全：启用
C T	history	历史列表：存储历史命令的最大数量
C T	home	home 目录
C T	ignoreeof	收到 eof 信号(^D)后不退出 shell
• T	implicited	只键入目录名意味着改变到这个目录
• T	listjobs	作业控制：列举所有挂起的作业； long =长格式
• T	loginsh	设置用来指示登录 shell
C T	mail	查看新电子邮件的文件列表
C T	noclobber	不允许重定向输出替换文件
C T	notify	作业控制：当后台作业结束时立即通知
• T	owd	最近[过去]的工作目录(与 cwd 相对)
C T	path	搜索程序的目录
C T	prompt	shell 提示(通过修改这个变量进行 shell 提示的定制)
• T	pushdsilent	目录栈： pushd 和 popd 不列举目录栈
• T	pushdtohome	目录栈：没有参数的 pushd 将假定 home 目录
• T	rmstar	强制用户在执行 rm * (删除所有文件)之前进行确认
• T	rprompt	屏幕右边的特殊提示(例如：设置成%~或者%/)
• T	savedirs	目录栈：在注销之前，保存目录栈
C T	savehist	历史列表：在注销之前，保存历史命令数量
C T	shell	登录 shell 的路径名
C T	term	正在使用的终端类型
C T	user	当前用户标识
C T	verbose	调试：只在历史替换之后回显每条命令
• T	visiblebell	使用屏闪替代发音

图 12-3 C-Shell 家族：最重要的 shell 变量

对于 C-Shell 家族来说，shell 使用了许多有特殊目的的 shell 变量。该表中所列举的变量是我认为最有用的。附录 G 中有更完整的 shell 变量列表。

最左边的一列说明哪个 shell 支持这个变量，其中 C=C-Shell，T=Tcsh。圆点表示相应的 shell 不支持该选项。

这又产生了最后一个问题。如果 C-Shell 家族使用 shell 变量控制 shell 的行为，那么 Bourne shell 家族使用什么来控制 shell 的行为呢？答案是一个称为“shell 选项”的精心制作的系统，我们将在本章后面详细讨论它。但是首先，我们需要讨论几个与变量使用相关的基本概念。

12.6 显示及使用变量的值：echo、print

使用 `env` 或 `printenv` 命令可以立即显示所有环境变量的值，而使用 `set` 命令可以显示所有的 shell 变量。但是，许多时候只希望显示一个变量的值。在这种情况下，可以使用 `echo` 命令。

`echo` 命令的任务就是显示赋予它的任何对象的值。例如，如果输入了：

```
echo I love Unix
```

那么将看到：

```
I love Unix
```

(到目前为止，这应该是正确的。)

为了显示一个变量的值，需要使用一个 `$`(美元符号)字符，后面跟着用花括号括起来的变量名。例如，显示变量 `TERM` 的值，可以输入：

```
echo ${TERM}
```

在自己的系统上试一试，看看得到什么结果。如果没有歧义，还可以省略掉花括号。

```
echo $TERM
```

大多数时候都是这样使用的，但是一会之后我将示范一个例子，这个例子需要使用花括号。

当我们谈论以这种方式使用变量时，一般将 `$` 字符读作“dollar”。因此，您可能听到有人说：“If you want to display the value of the `TERM` variable, use the command `echo-dollar-term`(如果您希望显示变量 `TERM` 的值，可以使用命令 `echo-dollar-term`)。”

表示法 `$NAME` 非常重要，因此我希望您记住它。当您只键入一个变量的名称时，它就是一个名称；当您键入一个 `$` 字符，后面跟一个名称时(如 `$TERM`)，它指的是该变量的值。因此，前面的 `echo` 命令意味着：“显示变量 `TERM` 的值。”

考虑下面的例子，这个例子与前面的例子相似，只是没有 `$` 字符。在这个例子中，`echo` 命令只显示字符串 `TERM`，并不显示变量 `TERM` 的值：

```
echo TERM
```

您可以使用 `echo` 命令以自己希望的任意方式显示变量和文本。例如，下面是一条显示终端类型的命令，但是该命令显示的信息更丰富：

```
echo The terminal type is $TERM
```

假设您的终端类型是 **xterm**, 那么您将看到:

```
The terminal type is xterm
```

在 shell 中, 一些标点符号字符称为“元字符”, 它们拥有特殊的含义(我们将在第 13 章中讨论)。为了防止 shell 解释元字符, 需要将元字符包含在双引号中。这就告诉 shell 照字面意义接受字符。例如, 为了显示尖括号中的 **TERM** 值。您可能输入下述命令:

```
echo The terminal type is <$TERM>.
```

但是, **<**和**>**字符是元字符, 表示“重定向”(参见第 15 章), 所以该命令不能正常运行(试试看)。为此, 您需要使用:

```
echo "The terminal type is <$TERM>."
```

提示

当使用 **echo** 命令显示标点符号时, 使用双引号告诉 shell 不要将标点符号解释为元字符。

在使用 **echo** 命令时拥有极大的灵活性。例如, 可以显示不止一个变量:

```
echo $HOME $TERM $PATH $SHELL
```

如果希望使用一个与它的邻居不分开变量, 则必须使用花括号为它划清界限。例如, 假设变量 **ACTIVITY** 的值为 **surf**。那么命令:

```
echo "My favorite sport is ${ACTIVITY}ing."
```

将显示:

```
My favorite sport is surfing.
```

提示

如果编写 shell 脚本, 将会发现必须大量地使用 **echo** 命令。花一点时间查阅一下说明书页(**man echo**), 说明书页中有许多选项和特性说明, 借助它们可以控制输出的格式和内容。

Korn shell 用户提示

所有的 shell 都允许使用 **echo** 命令显示文本和变量。至于 Korn shell, 还可以使用 **print** 命令:

```
print "The terminal type is $TERM."
```

Korn shell 的开发者 David Korn(参见第 11 章)创建了 **print** 命令来替换 **echo** 命令。他这样做是因为那时的两个主要 Unix 版本: System V 和 BSD(参见第 2 章)在使用 **echo** 命令时有一些细小的不同特性。这意味着使用 **echo** 的 shell 脚本并不总能从一个系统移植到另一个系统上。

为了解决这个问题, Korn 设计了 **print** 命令, 该命令在所有的系统上以相同的方式工作。现在, 这已经不再像 Korn 时代那样成为一个问题了。不过, 如果要编写 Korn shell 脚本, 而且还要在其他计算机上运行, 那么使用 **print** 命令取代 **echo** 命令时应该谨慎地决定。

12.7 Bourne shell 家族使用变量: export、unset

对于 Bourne shell 家族, 创建变量非常简单。所需做的全部事情就是键入一个名称, 后跟一个=(等号)字符, 再后跟一个值。变量的值必须是字符串。创建变量的语法为:

```
NAME=value
```

正如前面所述, 变量名可以使用字母、数字或者下划线(_)。但是, 变量名的第一个字符不能是数字。

下面举例说明, 您自己可以试一试(如果希望的话可以使用自己的名字)。一定要确保不在等号两边加空格:

```
HARLEY=cool
```

当以这种方式创建变量时, 我们就说您设置了变量。因此, 我们可以说上一个例子设置了变量 **HARLEY**, 并给变量 **HARLEY** 赋予一个值 **cool**。

如果希望使用一个包含有空白符(空格或者制表符, 参见第 10 章)的值, 则需要将值放在双引号中:

```
WEEDLY="a cool cat"
```

如果变量存在, 还可以使用相同的语法修改变量的值。例如, 一旦已经创建了变量 **HARLEY**, 那么使用下述命令就可以将该变量的值由 **cool** 修改为 **smart**:

```
HARLEY=smart
```

在 Bourne shell 家族中, 每个新变量都自动地被设置成 shell 变量(参见本章前面的讨论)。使用 **export** 命令可以将变量导出到环境中。导出时键入 **export**, 后面跟一个或者多个变量的名称。下面的例子导出变量 **HARLEY** 和 **WEEDLY**:

```
export HARLEY WEEDLY
```

HARLEY 和 **WEEDLY** 变量同时由 shell 变量变为 “shell+环境” 变量。

正如第 10 章中讨论的, 使用一个分号将各条命令分隔开, 可以在一个命令行上输入多条命令。因为创建一个变量, 然后立即导出这个变量非常常见, 所以一起输入两条命令非常有用, 例如:

```
PAGER=less; export PAGER
```

这种方式要比输入两条单独的命令快一些, 所以这种模式经常使用, 特别是在 Unix 文档资料和 shell 脚本中。但是, 还有一个更好的方法。**export** 命令实际上允许同时设置变量并导出到环境中。该命令的语法为:

```
export NAME[=value]...
```

下面举一个简单的例子:

```
export PAGER=less
```

仔细地看看该语法。注意 **export** 允许指定一个或多个变量名, 还可以为每个变量名指

定一个值。因此，使用一条命令就可以导出许多变量：

```
export HARLEY WEEDLY LITTLENIPPER
export PAGER=less EDITOR=vi PATH="/usr/local/bin:/usr/bin:/bin"
```

提示

通常，最出色的 Unix 用户脑筋转得比较快。同时，他们还喜欢采用尽可能简单的方式键入命令。基于这一原因，首选的方法就是用一条命令设置并导出变量：

```
export PAGER=less
```

尽管许多人使用两条命令来设置和导出变量，但是使用一条命令完成两份作业可以显示出您是一个聪明的人，与众不同的人。

正如前面所述，当创建变量时，我们称这是在设置变量。当删除变量时，我们称这是在复位(unset)变量。变量极少需要复位，但是如果需要的话，则可以使用 **unset** 命令。该命令的语法比较简单：

```
unset NAME...
```

下面举例说明：

```
unset HARLEY WEEDLY
```

提示

更有趣的是，在 Bourne shell 家族中，没有简单的方法从环境中移除变量。一旦变量导出，收回该变量的唯一方法就是复位该变量。

换句话说，从环境中移除一个 Bourne shell 变量的唯一方法就是销毁这个变量。

12.8 C-Shell 家族使用变量：setenv、unsetenv、set、unset

正如前面讨论的，与 Bourne shell 家族不同，C-Shell 家族在环境变量和 shell 变量之间有一个清晰的界限。换句话说，C-Shell 明确区分全局变量和局部变量。基于这一原因，在 C-Shell 家族中使用变量要比在 Bourne shell 家族中使用变量简单一些。

在 C-Shell 家族中，使用 **setenv** 和 **unsetenv** 命令可以设置(创建)或复位(删除)环境变量。设置或复位 shell 变量时，需要使用 **set** 和 **unset** 命令。

setenv 命令的语法如下所示：

```
setenv NAME [value]
```

其中 *NAME* 是变量名，*value* 是希望为变量设置的值。注意，该命令中未使用=(等号)字符。

下面举一些创建环境变量的例子。如果您希望体验一下，那么请记住，一旦创建了环境变量，就可以使用 **env** 或 **printenv** 命令显示它们。

```
setenv PATH /usr/local/bin:/usr/bin:/bin
setenv HARLEY cool
setenv WEEDLY "a cool cat"
setenv LITTLENIPPER
```

前 3 条命令分别设置一个变量并指定变量的值。在第三个例子中，使用了双引号，而将空白符(两个空格)包含进去。在最后一个例子中，指定了一个变量名 (**LITTLENIPPER**)，但是没有指定值。这样将创建一个值为 **null** 的变量。当只关心某个变量是否存在，但不关心它的值时，我们就可以这样做。

复位环境变量可以使用 **unsetenv** 命令。该命令的语法如下所示：

```
unsetenv NAME
```

其中 *NAME* 是变量名。

例如，复位(删除)变量 **HARLEY**，可以使用：

```
unsetenv HARLEY
```

设置 shell 变量可以使用 **set** 命令，该命令的语法如下所示：

```
set name[=value]
```

其中 *name* 是 shell 变量名，*value* 是希望设置给变量的值。

下面举例说明：

```
set term=vt100
set path=(/usr/bin /bin /usr/uch)
set ignoreeof
```

第一个例子很简单，只是将 shell 变量 **term** 的值设置为 **vt100**。

第二个例子描述了一个重点。当在 C-Shell 家族中使用变量时，使用圆括号将一组字符串括起来，而不是使用双引号。这样做实际上就是定义了一组可以单独访问的字符串。在这个例子中，**path** 的值被设置为圆括号中的 3 个字符串，

在最后一个例子中，我们指定了一个没有值的 shell 变量。该命令创建了一个值为 **null** 的变量。在这个例子中，变量 **ignoreeof** 存在就说明 shell 忽略 **eof** 信号。这将要求用户使用 **logout** 命令结束 shell(参见第 7 章)。

一旦 shell 变量存在，就可以使用 **unset** 命令删除这个 shell 变量。语法如下：

```
unset variable
```

其中 *variable* 是变量名。

下面举例说明。如果希望告诉 shell 关闭 **ignoreeof** 特性，可以使用：

```
unset ignoreeof
```

请确保自己理解了将变量设置为 **null** 和删除变量之间的区别。考虑下面 3 条命令：

```
set harley=cool
```

```
set harley
unset harley
```

第一条命令创建一个名为 **harley** 的 shell 变量, 并将该变量的值指定为 **cool**。第二条命令将变量 **harley** 的值设置为 **null**。最后一条命令完全删除变量 **harley**。

12.9 shell 选项: set -o、set +o

正如前面讨论的, 对于 C-Shell 家族来说, 可以使用 shell 变量控制 shell 行为的各个方面。对于 Bourne shell 家族来说, 则需要使用 **shell** 选项。例如, shell 是交互式的还是非交互式的就是由 shell 选项控制的。

shell 选项就像 on/off 开关一样。当打开一个选项时, 就说**设置**了这个选项。这将告诉 shell 以某种方式运行。当关闭这个选项时, 就说**复位**了这个选项。这也就是告诉 shell 停止以这种方式运行。

例如, shell 支持一个叫“作业控制”的功能, 允许在后台运行程序(我们将在第 26 章详细讨论)。打开作业控制需要设置 **monitor** 选项, 关闭作业控制需要复位 **monitor** 选项。默认情况下, 交互式的 shell 中 **monitor** 选项是打开的。

提示

单词“set”和“unset”根据讨论的是 shell 选项还是变量, 其含义有所不同。

shell 选项或者是 off 或者是 on, 它们不需要创建。因此, 当设置 shell 选项时, 就将 shell 选项打开。当复位 shell 选项时, 就将 shell 选项关闭。

变量就不同了。当设置变量时, 实际上是创建变量。当复位变量时, 实际上是永久地删除变量。

shell 选项设置或复位的方式有两种。第一种, 当 shell 启动时, 可以以普通的方式指定选项, 即为命令指定一个或者多个选项(参见第 10 章)。例如, 下述命令就是启动一个 Korn shell, 并设置(打开)**monitor** 选项:

```
ksh -m
```

除了标准的命令行选项外, 还有另外一种打开或关闭 shell 选项的方式, 即使用 **set** 命令的一种变体。下面给出该命令的语法。要设置一个选项, 可以使用:

```
set -o option
```

要复位一个选项, 可以使用:

```
set +o option
```

其中 *option* 是选项的“长名称”(参见图 12-4)。

例如, 假设 shell 正在运行, 您希望设置 **monitor** 选项, 则可以使用:

```
set -o monitor
```

为了复位 **monitor** 选项, 可以使用:

set +o monitor

在键入 **o** 时一定要小心，这是一个小写的字母“o”，而不是数字 0(只需记住，**o** 代表“option，选项”)。

刚开始时，使用**-o** 打开选项，而使用**+o** 关闭选项可能看上去比较奇怪。但是，我向您保证，最终您会认为这样是合理的*。

shell 每次启动时，根据 shell 是交互式的还是非交互式的，各种选项或者被默认设置，或者被默认复位。设计 shell 的程序员知道人们会如何使用 shell，因此在大多数情况下，默认 shell 选项就可以满足要求。这意味着极少需要修改 shell 选项。

但是，如果需要修改 shell 选项，则可以使用图 12-4 作为参考：该图中列出了交互式 shell 中最有用的 shell 选项。至于前面讨论的环境变量，如果不能完全理解，也不用担心。该列表只是用于参考。当您学习的有关 shell 选项使用的知识多了以后，您就会理解它们的作用。

shell	选项	长名称	含义
B K	-a	allexport	导出随后定义的所有变量和函数
B •	-B	braceexpand	启用括号扩展(生成字符模式)
B K	-E	emacs	命令行编辑器：Emacs 模式，关闭 vi 模式
B K	-h	hashall	查找到命令时(记住)的命令哈希位置
B •	-H	histexpand	历史列表：启用!风格替换
B •		history	历史列表：启用
B K	-I	ignoreeof	忽略 eof 信号^D；使用 exit 退出 shell(参见第 7 章)
• K		markdirs	在通配时，在目录名中追加/
B K	-m	monitor	作业控制：启用
B K	-C	noclobber	不允许重定向的输出替换某个文件
• K		nolog	历史列表：不保存函数定义
B K	-b	notify	作业控制：当后台作业结束时立即通知
• K		trackall	别名：为命令替换完整路径名
B K	-V	vi	命令行编辑器：vi 模式，关闭 Emacs 模式
• K		viraw	vi 模式：立即处理每个键入的字符

图 12-4 Bourne shell 家族：交互式 shell 选项汇总

这个表摘要汇总了交互式 shell 中有用的 shell 选项。更多的信息，请参见 shell 的说明书页。

最左边的一列说明哪个 shell 支持这个选项，其中 B=Bash，K=Korn Shell。圆点表示相应的 shell 不支持该选项。注意有一些选项，如 **history**，只有长名称，没有短的选项名称。

注：(1)尽管 Bash 支持 **emacs** 和 **vi** 选项，但是它不使用 **-E** 和 **-V**。(2)Korn Shell 使用 **-h** 选项，但是不支持长名称 **hashall**。

* 下面是一个简短的解释。从第 10 章知道，标准的选项形式是一个-(连字符)字符后面跟一个字母。在修改 shell 选项时，大多数时候都希望设置 shell 选项，也就是说打开 shell 选项，而不是复位 shell 选项。基于这一原因，人们将常见的语法(**-o**)用于“设置”，而不常见的语法(**+o**)用于“复位”。

随着 Unix 经验的日益丰富，您就会开始理解这种类型的理由。到那个时候，您的思维方式就会发生改变，可以更容易地使用 Unix(遗憾的是，同样的改变却更难满足女学生联谊会上的啦啦队队长)。

除了图 12-4 中说明的选项之外,还有许多其他的 shell 选项,这些选项大多数在非交互式的 shell 中 useful(也就是说,当编写 shell 脚本时)。另外,如果使用的是 Bash,那么有一个特殊的命令 **shopt**(shell options, shell 选项)可以用来访问更多的选项。

附录 G 中收录了完整的 shell 选项集,以及一些关于 **shopt** 的提示。尽管您现在还不需理解所有的内容,但是我希望您查看一下附录 G,从而知道有哪些选项可以使用。

提示

关于 shell 选项的权威信息,请参考 shell 的说明书页。

```
man bash
man ksh
```

对于 Bash,您需要搜索“SHELL BUILTIN COMMANDS”。对于 Korn shell,您需要搜索“Built-in Commands”或者“Special Commands”。

12.10 显示 shell 选项

Bourne shell 家族使用 shell 选项控制 shell 的操作。要显示 shell 选项的当前值,可以使用 **set -o** 或者 **set +o** 命令本身:

```
set -o
set +o
```

使用 **set -o** 将以一种容易阅读的方式显示所有 shell 选项的当前状态。使用 **set +o** 将以一种紧缩的格式显示相同信息,这种格式的信息适合用作 shell 脚本或者程序的数据。

如果输出对于屏幕来说太长,可以将输出发送给 **less**,从而每次一屏地显示输出信息:

```
set -o | less
set +o | less
```

如果想练习选项的设置和复位,可以试一试 **ignoreeof** 选项。正如第 7 章中讨论的,按下 **^D**(eof 键)可以终止 shell。但是,如果被终止的 shell 碰巧是登录 shell,那么系统将注销。

然而,人们很容易不小心按下 **^D** 键,从而在不经意间将自己注销。为了防止这种情况的出现,可以设置 **ignoreeof** 选项。这样就告诉 shell 在按下 **^D** 键时不结束 shell,必须输入 **exit** 或者 **logout** 命令才能结束 shell。设置该选项的命令如下:

```
set -o ignoreeof
```

复位该选项的命令如下:

```
set +o ignoreeof
```

您可以试一试设置、复位以及显示该选项。每次修改时,都显示该选项的当前状态,然后按下 **^D** 键看看发生什么情况。

提示

除非您是一名高级用户,否则您需要关心的选项只有 **ignoreeof**、**monitor** 和 **noclobber**, 以及 **emacs** 或者 **vi**。

monitor 选项用于启用作业控制,将在第 26 章讨论。**noclobber** 选项防止重定向标准输出时偶然移除文件(参见第 15 章)。**emacs** 和 **vi** 选项用来指定使用哪个内置的编辑器回调及编辑前面的命令。本书后面解释这一选项。

这些选项在环境文件中都已得到最佳的设置,该文件是每次启动新 shell 时自动执行的初始化文件。第 14 章中将讨论该文件。

名称含义

set

您注意到没有,就在本章一章中,我们以几种不同的方式使用了 **set** 命令,每种 **set** 命令都有自己的语法。我们使用 **set** 命令显示 shell 变量、创建 shell 变量、打开或关闭 shell 选项以及显示 shell 选项。

如果仔细地看一看,就会发现我们实际上处理了 4 个碰巧拥有相同名称的不同命令。显然,名称 **set** 有些特殊之处,从而使程序员乐于使用它。

这并没有听起来那么奇怪。您是否知道,在英语中,单词“set”拥有比其他任何单词更多的含义?查一下字典,我敢打赌您肯定会大吃一惊。

12.11 机器可读、人类可读

当程序以可以用作另一个程序的输入数据的方式显示复杂的输出时,我们就说该输出是**机器可读(machine-readable)**的。尽管这个术语想象机器人像人一样阅读,但是它其实指的是输出以适合于程序处理的方式格式化。例如,您有一个统计数据列表,各个数字通过逗号分隔,而不是组织成列。尽管您或我都难以阅读它,但是它适合作为程序的输入。

当输出设计得特别适于阅读时,我们就说该输出是**人类可读(human-readable)**的。这个术语并不经常使用,但是 GNU 实用工具的说明书页(参见第 2 章)中会出现这个术语。许多类型的 Unix 都使用 GNU 实用工具,包括 Linux。实际上,许多命令专门设计有生成人类可读输出的选项。

例如,在上一节中,提到过 **set -o** 命令以一种易于阅读的方式显示输出,而 **set +o** 命令显示的输出适合用作 shell 脚本的数据。另一种表述方法就是 **set -o** 命令生成人类可读的输出,而 **set +o** 命令生成机器可读的输出。

当然,并不是每个人都是相同的。即使一些内容适于人类阅读,也并不意味着您个人相对于适于机器阅读的输出来说,喜欢它更多一些。例如,我碰巧就更喜欢 **set +o** 的输出,而不是 **set -o** 的输出。

12.12 练习

1. 复习题

1. 交互式 shell 和非交互式 shell 之间有什么区别?
2. 环境是 shell 以及任何由 shell 启动的程序可用的一个变量表。环境中存储有什么类型的变量? 举 3 个例子。什么类型的变量不属于环境?
3. 在 Bourne shell 家族(Bash、Korn shell)中,使用什么命令使 shell 变量成为环境变量?
4. 如何显示所有环境变量的值? 您自己的 shell 变量? 一个单独的 shell 变量?
5. 解释术语“机器可读”和“人类可读”。

2. 应用题

1. 环境变量 **USER** 包含当前用户标识的名称。请示范 3 种显示该变量值的不同方法。
2. 创建一个环境变量 **SPORT**, 并指定它的值为“surfing”。显示这个变量的值。启动一个新 shell。再次显示这个变量的值, 以展示这个变量作为环境的一部分已经传递给新 shell。将 **SPORT** 的值修改为“running”, 然后显示新值。现在退出新 shell, 返回原来的 shell。显示 **SPORT** 的值。您看到了什么结果? 为什么呢?
3. 在 Bourne shell 家族(Bash、Korn shell)中, **ignoreeof** 选项告诉 shell 当按下[^]D(eof 键)时不进行注销。启动 Bash 或者 Korn shell。查看 **ignoreeof** 选项是打开的还是关闭的。如果 **ignoreeof** 选项是关闭的, 将它打开。按[^]D 键确认自己会不会被注销。然后, 关闭 **ignoreeof** 选项。再次按[^]D 键确认自己是否真的被注销了。

3. 思考题

1. 环境变量并不是真正的全局变量, 因为子进程对环境变量的修改并不能传递回父进程。假定情况不是这样, 说明黑客如何使用这个漏洞, 在他拥有账户的多用户系统上导致问题。
2. C-Shell 家族在环境变量和 shell 变量之间拥有明确的界限。Bourne shell 家族就没有这么明确, 因为一个新创建的变量, 默认情况下是 shell 变量, 而如果导出这个变量, 那么这个变量就既是局部变量又是环境变量。这两类系统的优缺点各是什么? 您认为哪类系统比较好? 为什么呢? 您认为 Bourne shell 家族为什么要以这样一种比较混乱的方式处理变量呢?

使用 shell：命令和定制

在第 11 章中，我们从总体上讨论了 shell。在第 12 章中，基于第 11 章的内容讨论了交互式 shell、进程、环境变量、shell 变量和 shell 选项。在本章中，我们将讨论其余的基本概念。

花时间阅读本章内容会给您很好的回报，原因有两点。首先，本章准备讨论的思想对熟练地使用 shell 非常重要。其次，当您开始将 shell 各方面的知识综合在一起时，将会发现本章即将讲述的技能将在日常工作中节省您大量的时间。

13.1 元字符

使用键盘可以键入字母、数字和一些其他字符，例如标点符号和算术符号。另外，还可以使用<Space>、<Tab>和<Return>键生成空格、制表符及新行字符(参见第 7 章)。

综合来讲，我们称这些字母和数字为字母数字字符(alphanumeric character)。使用字母数字字符比较直接：键入希望键入的，并且所见即所得。但是，当使用 shell 时，还有许多其他字符拥有特殊的含义。我们称这样的字符为元字符(metacharacter)，您需要明白它们的工作方式。

例如，正如第 10 章中讨论的，通过用分号将各条命令分隔开，可以在同一个命令行上输入多条命令。因为分号对 shell 拥有特殊的含义，所以它就是一个元字符。

一个更抽象的例子就是，当按下<Space>、<Tab>和<Return>键时，也使用了元字符。空格和制表符用作分隔命令不同部分的空白符(第 10 章)，而新行字符用作一行结束的标记(第 7 章)。

如果试图现在就记住所有元字符的作用，这将很难，因为记住的东西并没有意义。因为每个元字符都用来实现由 shell 提供的一个特殊服务，所以学习元字符的同时学习它提供的服务才有意义。

例如，前面解释过，当需要引用变量的值时要使用\$(美元)字符(例如\$TERM)。就这一点而论，\$就是一个元字符。但是，当我告诉您\$就是一个元字符时，您还不理解变量的含义，那么这对您又会有多大的意义呢？

下面我准备列举所有的元字符，但是我不期望您现在就能完全理解它们。后面将一个接一个地解释它们。现在，我只希望确保当您看到它们时知道它们是元字符，从而不会不小心地错误使用它们。

例如，如果您准备输入下述命令，shell 将会注意到分号，并把您键入的内容解释成两条命令：

```
echo The search path is $PATH; the shell is $SHELL.
```

但是，如果我们知道分号就是元字符，可以采用下述方式告诉 shell 不用管它：

```
echo "The search path is $PATH; the shell is $SHELL."
```

当我们以这种方式保护元字符时，我们就说“引用”了元字符。

我们稍后再讨论引用。在这之前，我希望先介绍一下完整的元字符列表。首先，请看图 13-1。该图显示了 Unix 使用的所有非字母数字字符，这些字符并不都是元字符。然后请看图 13-2，该图显示了各个元字符，并提供有元字符使用的简单描述。出于参考目的，该图中还列出了讨论每个特定元字符的具体章号。

字符	英文名称	Unix 绰号
&	ampersand(和号)	—
'	apostrophe(撇号)	引号、单引号
*	asterisk(星号)	星号
@	at sign(at 符号)	at
`	backquote(反引号)	反引号(backtick)
\	backslash(反斜线)	—
{ }	brace brackets(花括号)	花括号、波形括号
^	circumflex(音调符号)	插入记号
:	colon(冒号)	—
,	comma(逗号)	—
\$	dollar sign(美元符号)	美元
<Return>	enter、return(回车)	新行
=	equal sign(等号)	等号
!	exclamation mark(感叹号)	bang
>	greater-than sign(大于符号)	大于
-	hyphen(连字符)、minus sign(减号)	虚线、减号(参见本书提示)
<	less-than sign(小于号)	小于
#	number sign(序数符号)	hash、pound(参见本书提示)
()	parentheses(圆括号)	—
%	percent sign(百分比符号)	百分比
.	period(点号)	点
+	plus sign(加号)	加
?	question mark(问号)	—
"	quotation mark(引号)	双引号
;	semicolon(分号)	—

图 13-1 Unix 中使用的非字母数字字符

/	slash(斜线)	正斜线
<Space>	space(空格)	—
[]	square brackets(方括号)	方括号
<Tab>	tab(制表符)	—
~	tilde(波浪号)	—
_	underscore(下划线)	—
	vertical bar(竖线)	管道

图 13-1 (续)

本表列举了 Unix 中使用的全部非字母数字字符。作为 Unix 传统的一部分，大多数字符都有绰号，在讨论它们时会使用。例如，感叹号称为“bang”，\$TERM 读作“dollar term”，等等。如果您希望听起来像一个 Unix 专家，请使用 Unix 绰号。

字符	章号	名称	作用
{ }	24	花括号	花括号扩展：生成一种字符模式
	15	管道	命令行：创建一个管道线
<	15	小于	命令行：重定向输入
>	15	大于	命令行：重定向输出
()	15	圆括号	命令行：在子 shell 中运行命令
#	14	hash、pound	命令行：注释的开头，忽略该行其余部分
;	10	分号	命令行：用于分隔多条命令
`	13	反引号	命令行：命令替换
~	24	波浪号	文件名扩展：插入 home 目录的名称
?	24	问号	文件名扩展：匹配任意一个字符
[]	24	方括号	文件名扩展：与一组字符中的字符匹配
*	24	星号	文件名扩展：匹配 0 个或多个字符
!	13	bang	历史列表：事件标记
&	26	和号	作业控制：在后台运行命令
\	12	反斜线	引用：下一个字符转义
'	12	引号、单引号	引用：取消所有的替换
"	12	双引号	引用：取消大部分替换
{ }	12	花括号	变量：确定变量名称的界限
\$	12	美元符号	变量：用变量的值替换
<Return>	7	新行字符	空白符：标记一行结束
<Tab>	10	制表符	空白符：在命令行中分隔单词
<Space>	10	空格符	空白符：在命令行中分隔单词

图 13-2 shell 中使用的元字符

在 shell 中，许多非字母数字字符都拥有特殊的含义。这些字符都是元字符。本表示范了所有的元字符，以及相应的 Unix 绰号。最终，您将学会使用每个元字符的精确规则。注意花括号有两种不同的使用方式。

出于参考目的，本表同时还指出了讨论每个元字符的章号。

在图 13-2 中, 有几个元字符用于文件名扩展, 文件名扩展也称为“通配(globbering)”。通配定义了一整套使用元字符的规则, 第 24 章将对此展开详细讨论。

提示

图 13-1 中有两个非字母数字字符根据使用 Unix 的时间长短拥有不同的名称。这是因为在 20 世纪 90 年代中期 Linux 开始流行时, 惯例发生了变化。

连字符(-): 年青人称连字符为“dash”, 年长者称它为“minus”。

序数符号(#): 年青人称序数符号为“hash”, 年长者称它为“pound”。

如果您是一名学生, 那么当您与超过 30 岁的教授谈话时, 这非常重要。记住要说“minus”和“pound”, 这样显得对年长者比较尊重。

13.2 引用和转义

有时候, 可能希望按字面上的含义使用元字符, 而不使用其特殊的含义。例如, 将分号作为分号使用, 而不是一个命令分隔符。或者可能希望不按管道使用| (竖线)*。在这些情况中, 必须告诉 shell 按字面意义解释字符。这样做时, 可以称其为引用字符。

字符的引用方法有 3 种: 使用反斜线、使用一对单引号或者使用一对双引号。

引用元字符最直接的方法就是在元字符前面放一个反斜线(\)。这就告诉 shell 忽略反斜线之后的字符的任何特殊含义。例如:

```
echo It is warm and sunny\; come over and visit
```

在这个例子中, 我们在;(分号)字符前面放了一个反斜线。如果没有这个反斜线, 那么 shell 将把分号解释成一个元字符, 从而假定您输入了两条独立的命令: **echo** 和 **come**。当然, 这将生成错误。

提示

当希望引用单个字符时, 一定要使用反斜线(\), 而不是常规的斜线(/)。常规的斜线拥有完全不同的作用。它在路径名中使用(参见第 23 章)。

当使用反斜线引用单个字符时, 我们称反斜线为“转义字符(escape character)”。这是一个重要的概念, 因此我希望花一点时间讨论它。

您可能还记着, 当我们在第 6 章中讨论运行级别时, 提到了模式的概念。具体而言, 当计算机系统、程序或者设备可能是若干种状态之一时, 我们使用术语模式表示特定的状态。例如, 在第 6 章中, 我解释过 Unix 或者 Linux 系统可以引导到单用户模式或者多用户模式。与此类似, 当在第 22 章中讨论 vi 文本编辑器时, 您将会发现, 在任何时刻, vi 或者处于输入模式, 或者处于命令模式。

当程序位于一种特定模式, 而我们将它修改到另一种模式时, 称这是由一种模式转义

* 正如 Freud 曾经说过: “有时候竖线就是一个竖线。”

到另一种模式。当通过按下特定键修改模式时，称该特定键为转义字符^{*}。当对 shell 键入命令时，反斜线就是一个转义字符，因为它告诉 shell 从一种模式(注意元字符)改变到另一种模式(不用注意元字符)。希望大家记住转义字符的思想，因为今后会经常遇到它，特别是程序员。

在 Unix 中，单词“escape”有两种使用方法。最常见的就是我们所讨论的从一种模式转义到另一种模式。例如，当使用 vi 文本编辑器时，按下<Esc>键就可以从插入模式转义到命令模式。

在 shell 中，单词“escape”的使用稍微有所不同，它被用作引用的同义词。例如，考虑下面的例子：

```
echo It is warm and sunny\; come over and visit
```

在这个例子中，我们可以说使用反斜线转义了分号。这与使用反斜线引用分号的意思相同^{**}。

如果只是用来转义元字符，那么使用反斜线就足够了，因为我们可以同一行上使用不止一个反斜线。考虑下述命令：

```
echo It is warm (and sunny\); come over & visit
```

这条命令不能正确运行，因为这条命令中有 4 个元字符：(、)、;和&。为了使该命令运行，需要将这 4 个元字符转义：

```
echo It is warm \ (and sunny\)\; come over \& visit
```

现在这条命令就有效了，可以正常工作。但是，该命令不容易阅读。作为另一种选择方案，Unix 允许使用单引号引用一串字符。因此，上述命令可以修改为：

```
echo 'It is warm (and sunny); come over & visit'
```

在这个例子中，我们将单引号之间的所有内容引用。当然，这包括所有的字符，而不仅仅是元字符，但是它不影响字母数字式字符的显示(停一下，好好想想为什么会是这种情况)。

因此，到目前为止，我们有两种引用元字符的方法：使用反斜线引用单个字符，或者使用单引号引用一串字符。当需要时，可以在同一条命令中综合使用这两种类型的引用：

```
echo It is warm '(and sunny);' come over \& visit
```

大多数时候，反斜线和单引号引用已足够。但是，在有些情形中，使用第三种类型的引用即双引号引用更加方便。下面举例说明。

^{*} 现在您应该知道键盘上为什么有<Esc>或者<Escape>键了。设计它正是为了方便程序从一种模式转义到另一种模式。

^{**} 更精确地说，当讨论从一种模式转义到另一种模式时，“escape”用作不及物动词，也就是不作用于对象的动词。当讨论转义(引用)字符时，“escape”用作及物动词，也就是作用于对象的动词。

除非在甲壳虫乐队(Beatles)流行之前上的学，否则英语教师可能会忘记讲授语法。如果是这种情况，那么您可能不知道及物动词和不及物动词之间的区别，因此下面再举一个例子，从而引起您的好奇心。

当说“Feel the fabric and tell me if it is soft”时，使用的是及物动词。动词是“feel”，对象是“fabric”。

当说“Do you feel lucky?”时，使用的是不及物动词。动词是“feel”，没有作用对象(更准确地说，“feel”是一个系动词，而“lucky”是一个形容词，充当主语补足语)。

尽管难以相信，但是诸如此类的素材非常重要，特别是当您希望成为作家时。

有时候,可能希望在一个引用字符串中使用\$字符,用来引用变量的值。例如,下述命令在尖括号中显示用户标识和终端类型:

```
echo My userid is <$USER>; my terminal is <$TERM>
```

这种形式的命令不能正常工作,因为元字符<、;和>拥有特殊的含义(\$字符没有什么问题,我们希望它是元字符)。解决方法就是只引用那些我们希望取字面含义的元字符:

```
echo My userid is \<$USER\>; my terminal is \<$TERM\>
```

这条命令可行,但是它非常复杂。当然,我们可以使用单引号来取代反斜线:

```
echo 'My userid is <$USER>; my terminal is <$TERM>'
```

这样比较容易阅读,但是它引用了所有的元字符,包括\$。这意味着只能看到字面上的\$USER 和\$TERM,而不能看到这两个变量的值。对于这类情况,可以使用双引号,因为这样所有的\$元字符将保留它们特殊的含义。例如:

```
echo "My userid is <$USER>; my terminal is <$TERM>"
```

因为我们使用了双引号,所以<、;和>字符都被引用,但是\$字符没有被引用(自己试试看)。

除了\$字符之外,双引号还保留其他两个元字符的含义,这两个元字符是\ (反斜线)和` (反引号)。我们将在后面讨论反引号。现在,所需知道的就是它拥有的特殊含义在双引号中保留,但是不在单引号中保留。

下面是小结:

- 使用反斜线引用单个字符(称为转义了这个字符)。
- 使用单引号引用一串字符。
- 使用双引号引用一串字符,但是保留\$(美元)、`(反引号)和\ (反斜线)的特殊含义。

13.3 强引用和弱引用

从前面的讨论中可以看出,单引号比双引号的功能更为强大。基于这一原因,有时候称单引号为强引用(strong quote),双引号为弱引用(weak quote)。下面是记住它们的一种简单方法:因为单引号强,所以只需要一个符号;因为双引号弱,所以需要两个符号。

实际上,反斜线是所有引用中最强的一个(尽管我们没有给它一个特殊的名称)。反斜线可以引用任何东西,因此,如果单引号不起作用,可以试一试反斜线。例如,有一天您可能需要转义一个单引号:

```
echo Don\'t let gravity get you down
```

反斜线功能如此强大,所以它甚至可以引用新行字符(花点时间好好想想这一点)。

假设在某行的末尾键入了\

试试下面的例子，在每行的末尾按<Return>键：

```
echo This is a very, very long \
line that goes on and on for a \
very, very long time.
```

如果您这样做，将键入一条特别长的命令，而且将看到一行特别长的输出。

与反斜线不同，单引号和双引号不引用新行字符。情况就是这样的，您认为当在被单引号或者双引号引用的字符串中按下<Return>键时应该发生什么情况呢？例如，假设键入了下述内容：

```
echo 'This line ends without a second quote
```

当在这一行的末尾按<Return>键时，新行字符没有被引用，因此它还保留它的特殊含义，也就是说它还标识一行的结束。但是，这里有个问题，因为单引号没有匹配。

如果使用的是 C-Shell 或者 Tcsh，这时将得到一个错误消息，说明出现了一个未匹配的'字符。

如果使用的是 Bash 或者 Korn shell，那么 shell 处于等待状态，等待更多的键入，并且希望用户最终会键入第二个引号(Bourne shell 比 C-Shell 要乐观一些)。一旦键入了第二个引号，shell 就会将所有的内容组合成一个非常长的命令，命令中间含有新行字符。

作为练习，键入下述两行内容，看看 shell 如何处理：

```
echo 'This line ends without a matching quote
and here is the missing quote'
```

您能说明发生了什么情况，并解释为什么吗？

13.4 shell 内置命令：type

当输入命令时，shell 将命令分成不同部分，以进行分析。这种情况，我们称 shell 在解析命令。每条命令的第一部分都是命令的名称，其他部分是选项或参数(参见第 10 章)。

在解析命令之后，shell 决定如何处理命令，其可能性有两种。一些命令在 shell 的内部，这意味着 shell 可以直接解释它们。这些命令是内部命令，通常称为内置命令(builtin command，或者简称为 builtin)。其他所有命令是外部命令，即必须独自运行的独立程序。

当输入内置命令时，shell 在自己的进程内运行该命令。当输入外部命令时，shell 将搜索合适的程序，然后以一个单独的进程运行该命令。这一原则类似于，假如您打电话给一家大型公司请求客户服务。如果接电话的人可以回答您的问题，那么他就自己回答问题(内部命令)。否则，他将电话转给合适的人(外部命令)。^{*}

^{*} 下面从更精确的角度解释外部命令的处理方式。shell 创建一个子进程，然后等待。子进程开始执行实际程序。当程序终止时，子进程也将终止。然后控制将返回给父进程，从而导致子进程消失。接着父进程显示一个 shell 提示，邀请用户输入另一条命令(更多的信息，请参见第 12 章中有关进程的讨论。)

查看一条命令是不是 shell 内置命令的方法有两种。第一种，可以显示该命令的说明书页。外部命令都拥有自己的说明书页，而内置命令没有。内置命令或者记录在 shell 的说明书页中，或者记录在所有内置命令的一个特殊说明书页上。

一种查看某条命令是不是内置命令的更快捷的方式就是使用 **type** 命令。该命令的语法为：

```
type command...
```

例如：

```
type date time set
```

该命令的准确输出依赖于所使用的 shell。例如，下面是在 Bash 中使用该命令所获得的结果：

```
date is /bin/date
time is a shell keyword
set is a shell builtin
```

下面是在 Korn shell、Tcsh 和 C-Shell 中使用该命令所获得的结果：

```
date is a tracked alias for /bin/date
time is a reserved shell keyword
set is a special builtin
```

尽管输出有点不同，但是结果是相同的：**date** 是外部命令；其他命令是内置命令。

此时，理解“路径别名(tracked alias)”的含义并不重要。它只是一个技术上的不同，您可以忽略。同理，我们也不需要区分内置命令和关键字(Key word)：它们都在 shell 中内置(关键字是特殊的内部命令，用于编写 shell 脚本)。重要的事情是意识到 **date** 命令是一条外部命令，拥有自己的文件(一种系统上是 **/bin/date**，其他系统是 **/usr/bin/date**)。

Unix 和 Linux 系统差不多提供有数百条外部命令，但是有多少条内部命令呢？这取决于所使用的 shell。作为一个有趣的参考，图 13-3 列举了第 11 章中所讨论的 shell 的内置命令数量。

shell	内置命令数量
Bourne shell	18
Korn shell	47
C-shell	55
Bash	69
Tcsh	87
FreeBSD shell	97
Zsh	129

图 13-3 各种 shell 的内置命令数量

Unix 命令有数百种，其中大多数命令都是外部命令，也就是说，它们作为独立的程序存在。但是，每种 shell 都有特定数量的内部命令。内部命令是可以直接运行的命令。本图中的表格列出了每种 shell 中内置命令的数量，包括关键字。作为一个有趣的比较，本表中还包含了贝尔实验室早期 Unix 用户所使用的老版本 Bourne Shell 的内部命令数量。

13.5 学习内部命令

Unix 传统的一部分就是当有人创建了新工具时，他应该为其他用户提供该工具的文档资料。具体而言，就是期望当程序员编写新命令时，他能为这条命令提供一个说明书页。因为联机手册的格式已经定义好(参见第 9 章)，所以程序员在编程结束后不难创建说明书页。实际上，几乎所有的 Unix 程序在发行时都提供有说明书页，以作为官方文档资料。

对于外部命令来说这一系统特别适合。因为每条外部命令都是一个单独的程序，所以可以提供自己的说明书页。但是内部命令呢？正如前面讨论的，内置命令不是独立的程序，它们是 shell 的一部分。因为内置命令有许多条(参见图 13-3)，所以期望开发 shell 的程序员为每条内置命令开发一个单独的说明书页是不现实的。

实际上，所有的内置命令都记录在 shell 的说明书页中。例如，Korn shell 的内置命令就记录在 Korn shell 的说明书页中。因此，有关特定 shell 的内置命令的信息，需要查看合适的说明书页。下述几条命令可供使用：

```
man bash
man ksh
man tcsh
man csh
```

但是，一定要记住，shell 的说明书页非常长，可能需要搜索才能查找到所需的内容。

一些 Unix/Linux 系统对于内置命令拥有独立的说明书页。使用 **apropos** 命令可以查看系统是不是属于这种情况(参见第 9 章)：

```
apropos builtin
```

如果系统上有这样的说明书页，那么它就是快速查看所有内置命令列表的地方。对于 Linux 和 FreeBSD 来说，可以使用：

```
man builtin
```

对于 Solaris 来说，可以使用：

```
man shell_builtins
```

Linux 还有一个 **help** 命令，可以以若干种方式显示 **builtin** 说明书页中的信息。该命令的语法为：

```
help [-s] [command...]
```

其中 *command* 是命令的名称。

开始时，可以通过输入 **help** 命令本身显示一个所有内置命令的摘要列表。如果输出太长，则可以将输出发送给 **less**(第 21 章)每次一屏地显示信息：

```
help
help | less
```

这是显示所有内置命令的一个紧凑列表的命令，例如，当需要查找一条特定命令时。另外还可以使用 **help** 命令显示一条或多条具体命令的信息，例如：

```
help set
help pwd history kill
help help
```

可以看出，**help** 本身就是一条内置命令。

最后，如果只希望查看某条命令的语法，可以使用 **-s(syntax, 语法)** 选项：

```
help -s help
help -s pwd history kill
```

提示

当编写 shell 脚本时，可以使用特殊的内置命令 **for**、**if**、**while** 等来控制脚本流程。这些命令有时候称为关键字。

在使用 Bash 脚本时，查看关键字语法的最快捷方式就是使用 **help** 命令。例如，查看所有 Bash 关键字的语法，可以使用：

```
help -s case for function if select time while until
```

如果需要查看更多信息，则需要将 **-s** 选项忽略：

```
help case for function if select time while until | less
```

注意该命令中使用了 **less**，从而防止输出滚动出屏幕的范围。

13.6 外部命令及搜索路径

如果命令不是 shell 中内置的——大多数命令都不是内置的，那么 shell 必须查找出合适的程序来执行。例如，当输入 **date** 命令时，shell 必须查找 **date** 程序，然后运行它。因此，**date** 就是一条外部命令。

shell 如何知道在什么地方查找外部命令呢？shell 检查 **PATH** 环境变量(参见第 12 章)。与所有的变量一样，**PATH** 包含一串字符，这串字符就是一系列目录名称，我们称之为搜索路径。

我们将在第 24 章中详细讨论目录。现在，我希望您知道程序都存储在文件中，而每个文件都位于目录*之中。搜索路径是包含所有外部命令的程序的目录列表。因此，搜索路径中的一个目录中将包含存放 **date** 程序的文件。

如果希望查看搜索路径，那么显示 **PATH** 变量的值即可：

```
echo $PATH
```

* 您可以认为 Unix 目录同 Windows 或 Macintosh 的文件夹相似。但是，它们之间有细小但重要的区别。

下面是一些典型的输出：

```
/bin:/usr/bin:/usr/ucb:/usr/local/bin:/home/harley/bin
```

在这个例子中，搜索路径包含下述 5 个目录：

```
/bin
/usr/bin
/usr/ucb
/usr/local/bin
/home/harley/bin
```

您的系统中的搜索路径可能与此例有所不同，但是，在最大程度上，Unix 系统对存放外部命令的目录倾向于使用标准的名称。例如，我见过的每个 Unix 系统都有一个 **/bin** 和一个 **/usr/bin** 目录，同时许多系统有 **/usr/ucb** 目录。

当我们在第 24 章讨论了目录之后，就会理解这些名称。现在，只需了解名称 **bin** 用来指示一个存放程序的目录。

在我们的例子中，前 3 个目录 **/bin**、**/usr/bin** 和 **/usr/ucb** 存放系统中所有用户使用的程序。前两个目录位于所有的 Unix 系统上，并在 Unix 安装时自动设置。**/usr/ucb** 只位于一些系统上。它的任务是存放派生自伯克利 Unix(参见第 2 章)的程序。名称 **ucb** 是 University of California at Berkeley(加利福尼亚大学伯克利分校)的缩写。

接下来的两个目录是定制目录：**/usr/local/bin** 由系统管理员设置，用来存放为了本地使用而安装的程序；**/home/harley/bin** 指用户标识 **harley** 的 **home** 目录中一个名为 **bin** 的目录。用户可以为自己生成这样一个目录，使用它存放自己的程序。

当 shell 需要查找外部命令时，它在搜索路径中按指定的顺序逐个检查每个目录。在我们的例子中，shell 首先查看 **/bin** 目录。如果在这个目录中找不到期望的程序，那么 shell 将在 **/usr/bin** 目录中查找期望的程序。接下来，shell 沿着指定的顺序依次查找余下的各个目录。

当 shell 查找到期望的外部命令时，它就停止搜索并执行程序。如果没有一个目录包含期望的程序，那么 shell 将放弃搜索并显示一个错误消息。例如，如果输入命令 **weedly**，那么系统将显示一个类似于下面的消息：

```
weedly: command not found
```

13.7 修改搜索路径

在大多数系统上，一般情况下不必自己定义搜索路径，因为系统已经设置好 **PATH** 变量。但是，在特定环境中(稍后进行讨论)，可能需要修改搜索路径，下面将示范如何修改搜索路径。基本思想就是将修改 **PATH** 变量的命令放到登录时自动执行的初始化文件中(第 14 章将讨论初始化文件)。

首先，我们讨论如何将 **PATH** 变量设置成一个特定的值。我们将分别考虑 Bourne shell

家族和 C-Shell 家族的处理情况, 因为它们的命令有点不同。

对于 Bourne shell 家族(Bash、Korn shell)来说, 使用 **export** 命令(参见第 12 章)设置 **PATH**。使用 **export** 命令可以使 **PATH** 变量成为环境变量, 这意味着 shell 及所有随后生成的进程都可以使用它。下面是完成该任务的一条典型命令:

```
export PATH="/bin:/usr/bin:/usr/ucb:/usr/local/bin"
```

该命令本身非常直接: 将 **PATH** 的值设置成一个包含若干目录名称的字符串。可以看出, 各个名称之间用冒号隔开, 而且等号的两边没有空格。

为了自己设置 **PATH** 的值, 需要将这条命令(或者类似的命令)放到自己的“登录文件”中, “登录文件”是每次登录时自动执行的初始化文件。如果要修改搜索路径, 只需修改登录文件即可(第 14 章将进行详细介绍)。

对于 C-Shell 家族(C-Shell、Tcsh)来说, 需要使用一个稍微不同的命令, 因为设置的是 shell 变量 **path**, 而不是环境变量 **PATH**:

```
set path=(/bin /usr/bin /usr/ucb /usr/local/bin)
```

第 12 章中讲过, 无论何时, 当修改 **path** 时, shell 都会自动重置 **PATH**。因此, 对于 **PATH** 变量来说, 这条命令和前面的 Bourne shell 命令都生成相同的设置。

但是, 注意语法上的不同。在 Bourne shell 命令中, 我们将环境变量(**PATH**)的值设置成一个长的字符串。而在 C-Shell 命令中, 我们将 shell 变量(**path**)的值设置为一列名称。在 C-Shell 语法中, 一列名称包含一个或多个元素, 用空格隔开, 并且这一列元素用圆括号括起来。

刚才提到设置 **PATH** 环境变量的 Bourne shell 命令应该放在登录文件中。设置 **path** 的 C-Shell 命令应该放在“环境文件”中, 环境文件是一个不同的初始化文件, 在每次新 shell 启动时都会自动地执行(再次说明, 所有这些都将在第 14 章解释, 第 14 章中还将示范一些例子)。

在大多数系统上, 定义 **PATH** 变量的命令早已设置好, 因此不必使用刚才讨论的命令。但是, 您可能希望修改默认的搜索路径, 以适合自己使用。例如, 如果您编写自己的 shell 脚本和程序, 并将它们存放在自己的 **bin** 目录(**\$HOME/bin**)中, 那么您将希望把这个目录的名称添加到搜索路径中。

下面是两条展示如何完成这一工作的命令。第一条命令是针对 Bourne shell 家族的, 第二条命令针对 C-Shell 家族。

```
export PATH="$PATH:$HOME/bin"  
set path = ($path $HOME/bin)
```

注意命令的语法。该语法意味着“将搜索路径的值修改为旧值加上**\$HOME/bin**”。换句话说, 就是将名称**\$HOME/bin**追加到已有搜索路径的末尾。

另外, 您也有可能希望在已有搜索路径的开头插入一个新目录。命令如下:

```
export PATH="$HOME/bin:$PATH"  
set path = ($HOME/bin $path)
```

正如这些例子所示，可以基于变量的当前值修改变量的值。这是一个重要的理念，因此不要只是记住这种模式。要在此花一点时间，确保准确理解所有内容。

现在，既然已经知道了如何将自己的目录添加到搜索路径中去，那么这里有一个问题：应该把自己的目录放在列表的开头还是末尾呢？这取决于自己的意愿。

如果将自己的目录放在搜索路径的末尾，那么 shell 将最后检查该目录。如果将自己的目录放在搜索路径的开头，那么 shell 第一个检查它。例如，假设您编写了一个叫 **date** 的程序，并将它保存在 **\$HOME/bin** 中。现在您输入：

```
date
```

如果将自己的目录放在搜索路径的开头，shell 将运行您自己的 **date** 程序，而不是标准的 **date** 程序。通过这种方法，可以有效地将任何外部命令替换为自己的程序。另外，如果将自己的目录放在搜索路径的末尾，那么 shell 将运行标准的 **date** 程序，而不是您自己编写的。这样可以防止偶然用相同名称的文件替代程序。选择哪一种方式由您自己来决定。

作为一名程序员，可能还希望向搜索路径中添加另外一个目录。如果指定一个点(.)字符，将向搜索路径中添加自己的“工作目录”(工作目录就是当前正在工作的目录。我们将在第 24 章讨论)。

下面举一个可以帮助您澄清这个概念的例子。下述命令都将 **\$HOME/bin** 目录和工作目录添加到当前路径的末尾(且采取该顺序)。第一条命令针对 Bourne shell 家族，第二条命令针对 C-Shell 家族：

```
export PATH="$PATH:$HOME/bin:."
set path = ($path $HOME/bin .)
```

这样就告诉 shell，当查看程序时，最后一个要查看的目录就是当前工作的目录。

对搜索路径的详细讨论已经超出了本书的范围(而且并不是所有内容都是必需的)。通常，您只需使用已经默认设置好的搜索路径，在必要时进行微小的修改。

提示

除非您是一名专家，否则最好谨慎行事，将自己的目录放在搜索路径的末尾。

13.8 黑客如何使用搜索路径

将工作目录(.)放到搜索路径中比较方便，但是超级用户(**root**)或者其他拥有特殊权限的用户标识最好不要这样做。这样做可能会导致安全灾难，例如……

您是一名系统管理员，出于方便，您将工作目录放在了 **root** 的搜索路径的开头。有一名用户——其实他是一名黑客^{*}，请求您的帮助，因此您以 **root** 登录，切换到该用户的 home 目录(一种常见的情况)。然后输入 **ls** 命令列举用户目录中的内容(也是一种常见的情况)。

^{*}当然，我指的是一名坏黑客，也就是一名骇客，而不是一名好黑客。有关好黑客和坏黑客的讨论，请参见第 4 章。

您不知道的是这名黑客已经创建了另一个版本的 **ls** 程序, 将这一程序放在他的 **home** 目录中。这个伪造的 **ls** 程序就像真的 **ls** 程序一样, 但是当 **root** 账户运行它时, 它会产生一个副作用, 即创建一个拥有特殊权限的秘密文件。然后, 黑客可以使用这个文件(称之为后门)为他获取超级用户的访问权限。

下面是发生过程: 当您切换到用户的 **home** 目录时, 用户的 **home** 目录就成为您的工作目录。当输入 **ls** 命令时, **shell** 查看工作目录, 发现黑客版本的 **ls** 程序并运行它。接下来的事情您已经知道了, 黑客接管了系统, 您的生活将暴露无遗。

实际上, 这是以前黑客惯用的伎俩: 寻找一种以 **root** 账户运行已经被修改的程序的方法, 然后创建一个以后可以使用的后门。这个故事有什么教训呢? 那就是, 在修改任何用于系统管理的用户标识的搜索路径之前, 一定要好好地斟酌一番。

提示

确保所有系统管理员用户标识(包括 **root**)使用的搜索路径中没有包含工作目录, 或者用户可能访问的其他任何目录。

13.9 shell 提示

众所周知, 每当 **shell** 准备好接收输入命令时, 它就显示一个提示。您是不是期望可以修改这个提示? 实际上, 在提示显示上有广泛的自由, 而且一些人已经开发出一些精美的提示, 可以显示颜色以及各种不同类型的信息。我们先从简单的入手, 然后再讨论较复杂的定制。

最初, 所有的 **shell** 都是两个字符的提示: 一个字符后跟一个空格。Bourne **shell** 使用一个 **\$**(美元)字符, C-Shell 使用一个 **%**(百分比)字符。现在, 这一传统仍然保持。因此, 如果使用的 **shell** 是 Bourne **shell** 家族(Bash、Korn **shell**)中的一员, 那么最简单的 **shell** 提示为:

```
$ date
```

这里在提示之后键入了 **date** 命令, 以便显示出 **\$** 之后的空格。空格属于提示的一部分。如果使用的是 C-Shell 或者 Tcsh, 那么最简单的 **shell** 提示为:

```
% date
```

尽管 C-Shell 家族传统上指定 **%** 字符作为 **shell** 提示, 但是许多 Tcsh 用户使用 **>**(大于号)字符, 从而提醒用户该 **shell** 是一个增强版本的 C-Shell:

```
> date
```

最后一个惯例与超级用户相关。当以 **root** 登录时, 不管使用的是哪一种 **shell**, **shell** 提示总是一个 **#** 字符。这样做的目的就是提醒您是一名超级用户, 需要额外小心:

```
# date
```

在继续之前, 请看图 13-4, 该图汇总了各种基本的提示。这里只有少数几个约定, 因此我希望您能够记住它们的含义, 从而无论何时, 当看到 **shell** 提示时, 都可以立即回答下述两个问题: (1)您是以超级用户登录的吗? (2)如果不是, 那么您使用的是哪种类型的 **shell**?

shell	提 示
Bash	\$
Korn Shell	\$
C-Shell	%
Tcsh	%或者>
超级用户	#

图 13-4 标准 shell 提示

根据惯例，标准的 shell 提示由一个字符及其后面的一个空格组成。Bourne shell 家族使用一个\$(美元)字符，C-Shell 家族使用一个%(百分比)字符。唯一的例外就是 Tcsh，有时候它使用一个>(大于号)字符。当作为超级用户登录时，将看到一个#(hash)字符，不管使用的是哪一种 shell。

13.10 修改 shell 提示

正如上一节中所解释的，修改某个变量的值就能够修改 shell 提示。对于 Bourne shell 家族来说，需要修改一个名为 **PS1**^{*}的环境变量。对于 C-Shell 家族来说，需要修改一个名为 **prompt** 的 shell 变量。

我们从一个简单的例子开始。下面是一条命令，该命令适合于 Bourne shell 家族，将 shell 提示设置为一个\$(美元)字符后面跟一个空格：

```
export PS1="$ "
```

同样，下面是一条适合于 C-Shell 家族的命令，它使用标准的%(百分比)字符：

```
set prompt = "% "
```

如果是 Tcsh 用户，则可能习惯使用>(大于号)字符：

```
set prompt = "> "
```

在继续之前，请通过复习第 12 章中的基本概念，确保理解了这些命令。

变量有两种类型：全局变量和局部变量。全局变量称为“环境变量”。局部变量称为“shell 变量”。

所有的 Bourne shell 都将提示的值存储在一个叫 **PS1** 的环境变量中。使用 **export** 命令可以修改环境变量的值。因此，要学习上面的 **export** 命令。请注意该命令的语法，特别是=(等号)字符的前后都不能有空格。

所有的 C-Shell 都将提示的值存储在一个叫 **prompt** 的 shell 变量中。在 C-Shell 中，使用 **set** 命令修改 shell 变量的值。因此，要学习上面两条 **set** 命令。

此时，您可能奇怪，Bourne shell 使用环境变量(全局)存放提示，而 C-Shell 使用 shell(局部)变量存放提示，这样有意义吗？通常，这并不重要，当修改 shell 提示时，只要使用合

^{*} 名称 **PS1** 意味着“prompt for the shell, number 1(shell 提示，编号 1)”。这样的变量还有 3 个，分别是 **PS2**、**PS2** 和 **PS4**，但是不需要修改它们，因此不用关心它们。详细内容请参见 shell 的说明书页。

适的命令(**export** 或者 **set**)即可。但是，它们之间的区别在讨论初始化文件时非常重要，初始化文件是每次登录时帮助自动设置提示的文件。第 14 章将详细讨论初始化文件。

到目前为止，我们只是对 shell 提示进行简单的修改。但是，通过操纵合适的变量，可以按照自己的希望任意设置 shell 提示。例如，可以设置 shell 提示显示一个有趣的消息：

```
export PS1="Enter a command please, Harley$ "
set prompt = "Enter a command please Harley% "
```

实际上，有趣的 shell 提示会很快失去吸引力。下面是一些更有用的提示：在提示中显示 shell 的名称(它适合于使用自己的特定 shell 的情形)。

```
export PS1="bash$ "
export PS1="ksh$ "
set prompt = "csh% "
set prompt = "tcsh> "
```

下面是这些命令生成的提示：

```
bash$
ksh$
csh%
tcsh>
```

这种类型的提示对于超级用户特别方便。例如，假设用户标识 **root** 使用 **Bash** 作为默认 shell。如果设置 shell 提示如下：

```
export PS1="bash# "
```

那么提示将为：

```
bash#
```

#号将提醒您是一名超级用户，而名称提醒您正在使用哪一种 shell。

除了使用单词和字符，还有其他 3 种方法来增强 shell 提示。您可以：

- 将变量的值插入到提示中。
- 利用转义字符使用各种特殊码。
- 将命令的结果插入到提示中(这就是所谓的命令替换)。

以上每种技巧都有自己的重要性，而且其价值也远远不止修改 shell 提示。基于这一原因，下面我们分别讨论各种思想，从而使您能够理解普遍原则。

13.11 使用变量的值

正如第 12 章中所讨论的，为了使用变量的值，可以键入一个\$(美元)字符，后面跟用花括号括起来的变量名称。例如：

```
echo "My userid is ${USER}."
```

出于方便考虑，如果变量名与其他字符之间不需要分隔，可以省略掉花括号。例如：

```

echo "My userid is $USER."
```

提示

当使用变量的值时，使用花括号是一种好习惯，即便不是必需。这样做是为了增强命令的可读性，特别是在 shell 脚本中。此外，因为花括号将变量隔离，所以它们还可以帮助避免难以理解的语法问题，否则可能会导致极大的麻烦。

在 shell 提示中使用变量的值相当直接。例如，要将用户标识插入到提示中，可以使用：

```

export PS1="${USER}$ "
set prompt = "${USER}% "
```

(第一条命令针对 Bourne shell，第二条命令针对 C-Shell。)

如果用户标识是 harley(一种表现个性的方式)，这两条命令将生成如下所示的提示：

```

harley$
harley%
```

您希望在 shell 提示中使用哪个环境变量呢？第 12 章的图 12-2 中列举了一些最重要的环境变量。原则上，可以使用任何变量。但是，大多数环境变量并不适合在 shell 提示中使用。为了帮助缩小选择范围，图 13-5 列举了我认为最重要或者最有趣的 shell 提示。如果希望体验一下，只需从下述命令中选择一个适合自己 shell 的命令，然后将变量替换为图 13-5 中的变量即可。

```

export PS1="${VARIABLE}$ "
set prompt = "${VARIABLE}% "
```

shell	变量	含义
B K C T	HOME	home 目录
. . . T	HOST	计算机名称
B . . .	HOSTNAME	计算机名称
B . . T	HOSTTYPE	主机计算机类型
B K C T	LOGNAME	当前用户标识
B K C T	PWD	工作(当前)目录
B K . .	RANDOM	0 至 32 767 之间的随机数
B K . .	SECONDS	当前 shell 的运行时间(单位为秒)
B K C T	SHELL	登录 shell 的路径名
B K C T	USER	当前用户标识

图 13-5 在 shell 提示中有用的环境变量

增强 shell 提示的一种方法就是在 shell 提示中包含有用或者有趣的变量的值。第 12 章中的图 12-2 包含了所有重要的环境变量。本表只显示适合于 shell 提示的环境变量。

最左边的一列说明哪个 shell 支持这个变量：B=Bash；K=Korn Shell；C=C-Shell；T=Tcsh。圆点表示相应的 shell 不支持该选项。

大多数人喜欢使用 **LOGNAME**、**PWD**、**SHELL** 或者 **USER**。但是, 对我而言, 可以一直看个不停的最有趣的变量是 **RANDOM** 和 **SECONDS**。但是, 只有 **Bash** 和 **Korn shell** 中才有这两种变量。下面是体验这两个变量的命令:

```
export PS1='Your lucky number is ${RANDOM} $ '
export PS1='Working time: ${SECONDS} seconds $ '
```

13.12 引用变量时使用哪类引号

仔细看一看上一节中的两个例子:

```
export PS1='Your lucky number is ${RANDOM} $ '
export PS1="${USER}$ "
```

您是否注意到, 其中有一条命令使用单引号, 而另一条命令使用双引号? 这一区别描述了一个细小但是重要的知识点, 希望您能够理解, 特别是在您打算编写 **shell** 脚本时。

使用两种不同类型引号的原因在于所使用的两个变量, 其中一个变量发生变化, 而另一个变量不发生变化。更准确地说, 就是 **RANDOM** 的值是一个随机数, 每次查看它时都不相同。而 **USER** 的值是用户标识, 总是保持不变。

在引用 **\${USER}** 时使用的是双引号, 从而允许将 **\$** 字符解释成元字符。这意味着在处理该命令时 **USER** 的值就固定下来了, 这样没有问题, 因为 **USER** 的值永远不发生改变。

在引用 **\${RANDOM}** 时使用的是单引号, 从而允许保留 **\$** 字符的含义, 以便于稍后使用。这一技术确保直至创建实际提示时才对 **RANDOM** 求值。通过这种方式, 当要显示一个新的提示时, **shell** 就使用这个时刻 **RANDOM** 所拥有的值。

此时, 回想一下本章前面讨论的强引用和弱引用或许会有帮助。总之:

- 单引号('...'), 也称为强引用, 引用所有内容。在单引号中, 没有字符拥有特殊的含义。
- 双引号("..."), 也称为弱引用, 除 3 个元字符 **\$**(美元)、**`**(反引号)和 ****(反斜线)外引用所有内容。在双引号中, 这 3 个字符还保留它们各自的特殊含义。

因此, 当在命令中使用 **'\${VARIABLE}'** 时, 所有的字符都取字面上的含义, **\$** 字符保留下来, 以便稍后使用其原本含义。当使用 **"\${VARIABLE}"** 时, **\$** 字符就被解释成元字符, 而整个表达式在这个时刻就被替换为 **VARIABLE** 的值。

因此, 当需要引用变量时, 心里要想一想: “该变量的值在使用之前是否会变化?” 如果答案是肯定的, 则使用强引用(也就是单引号)来防止 **\$** 字符被解释, 直至需要它们。否则, 使用弱引用(双引号), 从而允许 **\$** 字符被立即解释。

13.13 使用转义字符的特殊码

到目前为止, 已经解释过 **shell** 提示可以包含任何希望的字符, 以及一个或多个变量的

值。在本节中，将讨论如何使用特殊码来增强 shell 提示。

在讨论过的 4 种 shell 中，只有 Bash 和 Tcsh 允许使用这样的特殊码。这些特殊码允许在 shell 中插入各种类型的信息：工作目录的名称、用户标识、计算机的主机名等。如果不嫌麻烦，还可以在 shell 中集成颜色、下划线以及粗体字，但是大多数人嫌麻烦，不愿意这样做。

基于参考目的，图 13-6 示范了最有用的特殊码。这些特殊码的完整列表位于 shell 的说明书页中。如果您喜欢进取，而且有额外的时间，那么您可以学习在 shell 提示中如何使用颜色和其他效果。如果是这样，也可以在 Web 上搜索帮助信息(提示：使用这样的特殊码相当复杂，因此如果现在还不能理解，请不用担心)。

含义	Bash	Tcsh	Korn shell	C-Shell
工作目录：~表示法	\w	%~	.	.
工作目录：只有基名	\W	.	.	.
工作目录：完整路径名	.	%/	SPWD	.
计算机的主机名	\h	%m	`hostname`	`hostname`
当前用户标识	\u	%n	\$LOGNAME	\$LOGNAME
shell 的名称	\s	.	`basename \$SHELL`	`basename \$SHELL`
时间：AM/PM 表示法	\@	%@	.	.
时间：24 小时制表示法	\A	%T	.	.
日期	\d	.	.	.
星期几	.	%d	.	.
月的第几天	.	%D	.	.
月	.	%w	.	.
年	.	%Y	.	.
历史列表：事件编号	!\	%!	!	!

图 13-6 shell 提示中使用的特殊码、命令和变量

Bash 和 Tcsh 允许使用特殊码在 shell 提示中插入信息。本表列出了最有用的特殊码。有关特殊码的完整列表，请参见 Bash 或 Tcsh 的说明书页。圆点表示相应的 shell 不支持该选项。

Korn shell 和 C-Shell 不支持特殊码。但是，为了全面地考虑问题，本表也示范了使用变量来模拟少数几个特殊码的方式。

图 13-6 中每个 Bash 和 Tcsh 码都包含一个转义字符，后面跟另一个字符(正如本章前面讨论的，转义字符告诉 shell 从一种模式改变到另一种模式)。对于 Bash 来说，shell 提示的转义字符是\ (反斜线)；对于 Tcsh 来说，shell 提示的转义字符是%(百分号)。

正如所述，只有 Bash 和 Tcsh 使用特殊码。但是，其他 shell 也需要在 shell 提示中放置这样的信息。在少数情况下，可以使用命令和变量实现，图 13-6 中也示范了这些内容(与图 13-5 进行比较)。这些变量有的在包含反引号的表达式中使用。其语法稍后在命令替换一节中解释。

大多数 shell 提示比较容易理解。但是,下面将提及的关于工作目录的特殊码要在理解了目录(参见第 24 章)之后才有意义,而关于历史事件编号的特殊码要在学习了使用历史列表(本章后面讨论)之后才有意义。

为了说明这些特殊码的工作方式,下面举几个例子。要将用户标识插入到提示中,可以使用 `\u`(Bash)或 `%n`(Tcsh)。因此,下面两条 Bash 命令拥有相同的效果:

```
export PS1="\u$ "
export PS1="${USER}$ "
```

如果用户标识是 **harley**, 那么提示是:

```
harley$
```

同理,下述两条 Tcsh 命令拥有相同的效果:

```
set prompt = "%n> "
set prompt = "${USER}> "
```

图 13-6 中的特殊码非常直观,当有时间时,您可以自由地体验它们。如果喜欢,还可以同时使用不止一个特殊码。例如,在 Bash 提示中显示日期和时间,可以使用:

```
export PS1="\d \@ $ "
```

在 Tcsh 提示中,可以使用:

```
set prompt = "%d %w %D %e> "
```

13.14 命令替换

在本节中,我们将讨论 shell 提供的一个最神奇、最强大的特性:命令替换(command substitution)。命令替换允许在一条命令中嵌入另一条命令。shell 首先执行嵌入的命令,并且用输出替换该命令。然后 shell 再执行整个命令。

很明显,这里涉及的是一个复杂的思想,因此我首先从几个例子入手。然后,再示范一个实际应用:如何在 shell 提示中使用命令替换显示有用的信息,且如果不这样做则无法在提示中显示这样的信息。

我们首先从基本的语法开始。通过将一条命令封装在 ```(反引号)字符中,可以将它嵌入到另一条命令中。例如:

```
echo "The time and date are `date`."
```

在这个例子中, **date** 命令就封装在反引号中。这样就告诉 shell 用两步执行整个命令。首先,求 **date** 命令的值,并将 **date** 命令的输出替换到较大的命令中。然后,执行较大的命令(在这个例子中是 **echo**)。

假设现在是 2008 年 12 月 21 日,星期一上午 10:30,并且位于美国太平洋时区。

那么 **date** 命令的输出为:

```
Mon Dec 21 10:30:00 PST 2008
```

在第一步中, shell 用上述值替换 **date** 命令, 将原始命令修改为:

```
echo "The time and date are Mon Dec 21 10:30:00 PST 2008."
```

在第二步中, shell 执行修改后的 **echo** 命令, 生成下述输出:

```
The time and date are Mon Dec 21 10:30:00 PST 2008.
```

尽管我将解释分成了两部分, 但是这些发生得非常快, 对我们而言, 就好像 shell 立即显示最终结果一样。

下面再举一个例子。环境变量 **\$SHELL** 包含有 shell 程序的路径名。假设使用的 shell 是 Bash, 如果输入:

```
echo $SHELL
```

将得到下述输出(或者类似的内容):

```
/bin/bash
```

这意味着您使用的 shell 是 **bash** 程序, 它位于 **/bin** 目录中(我们将在第 24 章中讨论目录和路径名)。

完整的表达式 **/bin/bash** 称为路径名。但是, 在这个例子中, 我们不关心整个路径名, 只需要其最后一部分(**bash**)。使用 **basename** 可以抽取任何路径名的最后一部分, 它的目的就是读取一个完整的路径名, 并输出路径名的最后一部分。例如, 下述命令的输出是 **bash**:

```
basename /bin/bash
```

更一般地讲, 为了显示 shell 程序的名称而不显示路径名的其余部分, 可以使用:

```
basename $SHELL
```

现在考虑如何通过命令替换将上述命令用作 shell 提示的一部分。假设希望将 shell 的名称作为提示的一部分显示。那么所需做的全部工作就是将 **basename** 命令的输出插入到设置提示的命令中:

```
export PS1="`basename ${SHELL}`$ "
set prompt = "`basename ${SHELL}`% "
set prompt = "`basename ${SHELL}`> "
```

第一条命令针对 Bash 或 Korn shell, 第二条命令针对 C-Shell, 第三条命令针对 Tcsh。

可以看出, 命令替换用来创建不使用命令替换就无法提供的功能。例如, 在上一节中, 我们讨论了使用特殊码将信息插入到 shell 提示中。但是, 只有 Bash 和 Tcsh 才可以使用这些特殊码。那么其他 shell 该怎么办呢? 解决方法就是使用命令替换。

例如, 从图 13-6 中可以知道, 将主机名(计算机名称)插入到 shell 提示的特殊码是

\h(Bash)和%m(Tcsh)。对于其他 shell, 可以使用命令替换。

基本的方法总是相同的。首先是回答一个问题: 哪条命令进行第一部分作业? 然后推断将这条命令的输出替换到另一条命令(进行第二部分作业的命令)中的最佳方法。

在这个例子中, 应该想一想: 哪条命令显示计算机的名称? 答案就是 **hostname**(参见第 8 章)。更具体地讲, 根据所使用 Unix 或者 Linux 的版本, 可能需要也可能不需要 **-s** 选项。要查看哪种变体比较适合于自己的系统, 可以使用命令:

```
hostname
hostname -s
```

现在只需要将 **hostname** 的输出替换到设置提示的命令中(如果不需要, 可以省略 **-s** 选项)。

```
export PS1="`hostname -s`$ "
set prompt = "`hostname -s`% "
```

其中第一条命令适合于 Bourne shell 家族(Bash、Korn shell), 第二条命令适合于 C-Shell 家族(C-Shell、Tcsh)。

最后一个例子。与 Bash 和 Tcsh 拥有在 shell 提示中显示主机名的特殊码相同, 它们也拥有显示用户标识的特殊码(分别是 **\u** 和 **%n**)。但是, 您已经知道, 其他 shell 中没有这样的特殊码。

一种解决方法就是如本章前面所做的那样, 在 shell 提示中使用 **\$USER** 变量。另外, 还可以对 **whoami** 命令(参见第 8 章)使用命令替换:

```
export PS1="`whoami`$ "
set prompt = "`whoami`% "
```

作为本节的小结, 下面再次强调前面讨论过的一些事情。当使用单引号(强引用)时, 两个引号之间的任何东西都无法保留特殊的含义。当使用双引号(弱引用)时, 只有 3 个元字符 **\$**(美元)、**`**(反引号)和 ****(反斜线)保留特殊的含义。现在您应该理解该列表中为什么要包含反引号了。

提示

反引号字符只在命令替换中使用。在使用时一定要小心不要与单引号或双引号混淆。尽管名称和外观与引号有些相似, 但反引号实际上与引用无关。这或许就是许多 Unix 用户使用名称 “backtick(反引号)” 替换 “backquote(反引号)” 的原因。

13.15 键入命令并进行修改

一旦使用过 Unix 一段时间, 就会知道因为需要进行一个微小的修改而不得不再次键入整个命令是多么令人灰心。特别是容易犯拼写错误(就像我一样)时情况更加烦人。为了方便起见, shell 提供了几个特性, 从而使命令的输入大为简化, 这些特性包括: 历史列表、命令行编辑、自动补全和别名。在本章的剩余部分, 我们将分别讨论这些特征, 每节讨论

一个特性。由于细节方面会根据 shell 的不同而有所不同，因此当需要帮助时，shell 的说明书页才是最终的权威参考指南。

自从 shell 面世以来，人们就开始争论哪种 shell 是最好的。依我的观点来看，您喜欢什么样的 shell 特性都无所谓，我个人认为那些键入(或者重新键入)命令最方便的 shell 就是最好的。通常，最好的特性只在 Bash 和 Tcsh 中可用，这就是为什么有经验的用户喜欢选择这两种 shell 的原因。

首先，我们回顾一下第 7 章中讨论的一些与错误纠正及命令行编辑相关的思想。然后再学习新的知识：

- 使用<Backspace>键可以删除键入的最后一个字符。这个键将发送 **erase** 信号(对于某些类型的计算机来说——例如 Macintosh，需要使用<Delete>键，而不是<Backspace>键。)
- 使用[^]W(<Ctrl-W>)键可以删除键入的最后一个单词，这个键发送 **werase** 信号。
- 使用[^]X 或[^]U 键(取决于系统类型)可以删除整行，这个键发送 **kill** 信号。
- 使用 **stty** 命令可以显示系统上所有的键映射。

对于大多数(并不是全部)shell 来说，还可以使用<Left>和<Right>光标控制键在命令行中移动。例如，假设您打算输入命令 **date**，但是却键入了：

```
dakte
```

这时光标位于命令行的末尾。只需按<Left>键两次，就可以将光标向左移动两个位置。然后按<Backspace>键删除字符 **k**，就可以按<Return>键输入命令了。注意<Return>键可以在命令行的任意位置上按下，不必仅限于命令行的末尾。

除了修改当前的命令行之外，还可以使用<Up>键调取上一条命令，然后对这条命令进行修改并重新输入。按<Up>键多次可以查找更早的命令，而按<Down>键可以再返回来查找最近的命令。

在 Bash 和 Tcsh 中可以采用这种方式使用<Left>、<Right>、<Up>和<Down>键，但是在 Korn shell 和 C-Shell 中并不能这样。使用 Bash 时，还有一个好处，即不仅可以使<Backspace>键删除左边的一个字符，还可以使用<Delete>键删除右边的一个字符。一旦习惯了这一点就会获得极大的便利(如果系统不支持<Delete>键，则可以使用[^]D)。

提示

使用<Left>和<Right>键在命令行中移动，使用<Up>和<Down>键调取前面的命令都非常方便，因此我建议您多加练习，直至不假思索就可以使用这些键。

在输入命令时，您的格言应该是：重用，而不重新键入。

13.16 历史列表：fc、history

在输入命令时，shell 会将命令保存到所谓的历史列表中。您可以采用不同方式访问历史列表，调取前面的命令，然后再对命令进行修改，并重新输入命令。例如，当按<Up>

和<Down>键调用命令时，实际上是在历史列表中向后和向前移动。

但是，<Up>和<Down>键只允许每次查看一条命令。还有一个功能更强大的特性，它允许您显示全部或者部分历史列表，然后再选择一条特定的命令。这种方法的工作方式取决于使用的 shell。通常，Bourne shell 家族(Bash、Korn shell)使用 **fc** 命令，而 C-Shell 家族(C-Shell、Tcsh)使用 **history** 和 **!**命令。大多数人发现 C-Shell 系统比较容易，基于这一原因，Bash 允许使用任何一种命令体系。下面介绍详细情况。

在历史列表中，每条命令称为一个事件，而每个事件都有一个内部编号，称为事件编号。历史列表的功能就是它可以基于事件编号调取命令。例如，可以告诉 shell 调取命令#24。

对于 Bourne shell 家族，可以使用 **fc** 命令加上 **-l(list, 列举)**选项显示历史列表(稍后再解释 **fc** 命令)。

```
fc -l
```

对于 C-Shell 家族，可以使用 **history** 命令：

```
history
```

这两条命令的输出都是每行一个事件，事件前面有事件编号。事件编号并不是命令的一部分，显示它们只是为了方便。下面举例说明：

```
20 cp tempfile backup
21 ls
22 who
23 datq
24 date
25 vi tempfile
26 history
```

如果历史列表太长，以至于滚动出了屏幕边界，可以使用 **less** 命令：

```
history | less
```

注意输入的每条命令都会添加到历史列表中，包括有错误的命令，以及 **history** 和 **fc** 命令本身。

通过事件编号也可以调取并执行特定的命令。对于 Bourne shell，可以使用加 **-s(substitute, 替换)**选项的 **fc** 命令，后面跟事件编号。例如，假设希望重新执行编号为 24 的命令，可以使用：

```
fc -s 24
```

对于 C-Shell 来说，该命令更加容易。只需键入一个 **!(bang)**字符，后面跟事件编号即可。注意！之后不能有空格：

```
!24
```

还有一种特殊的情况，即希望重复输入之前输入的最后一命令。对于 Bourne shell 来说，使用 **fs -s**(不必使用事件编号)就可以重新执行上一条命令：

```
fc -s
```

对于 C-Shell, 可以连续键入两个 **!** 字符:

```
!!
```

两种 shell 都允许在重新执行命令之前对命令进行小的修改。对于 **fc** 命令来说, 语法如下:

```
fc -s pattern=replacement number
```

对于 C-Shell 家族来说, 语法为:

```
!number:s/pattern/replacement/
```

在两种情况中, *pattern* 和 *replacement* 指的都是字符串, 而 *number* 指的是事件编号。

例如, 在上一个例子中, 事件编号 25 的命令是一条启动 **vi** 编辑器并打开一个叫 **tempfile** 文件的命令:

```
25 vi tempfile
```

假设希望再次运行这条命令, 但是这次希望打开一个叫 **data** 的文件。也就是说, 希望重新调用事件编号 25 的命令, 将 **tempfile** 文件修改为 **data** 文件, 然后再执行这条命令。对于 Bourne shell, 可以使用:

```
fc -s tempfile=data 25
```

对于 C-Shell, 可以使用:

```
!25:s/tempfile/data/
```

再次说明, 如果希望使用最近用过的命令, 那么命令就比较简单。例如, 假设您希望运行 **date** 命令, 但是不小心输成了 **datq**, 这样将显示一个错误消息:

```
$ datq  
datq: Command not found.
```

因此您希望将 **q** 修改成 **e**, 然后再重新执行这条命令。对于 **fc -s** 来说, 如果不指定事件编号, 则默认为上一条命令:

```
fc -s q=e
```

对于 C-Shell, 可以使用下述语法:

```
^pattern^replacement
```

例如:

```
^q^e
```

我知道它看上去比较奇怪, 但是它比较快速和方便, 您今后将会大量地使用它, 特别

是遇到需要进行微小修改的长命令时。例如，假设您希望复制 **masterdata** 文件，并将副本命名为 **backup**。这时可以使用 **cp** 命令(参见第 25 章)，但是您键入了：

```
cp masterxata backup
```

您得到了一个错误消息，因为当您键入第一个文件名时，不小心键入了 **x**，而不是 **d**。为了修复这个错误并重新运行该命令，只需输入下述命令即可：

```
^x^d
```

提示

当使用 Bash 时，相对于其他 shell，它有两个重要的优点。

首先，对于历史列表命令，Bash 拥有全世界最好的命令。您既可以使用 **fc** 命令，也可以使用 **history/!** 系统，可以根据自己的意愿选择。

如果不确定使用哪一种方式，可以先使用 **history/!** 系统。

其次，Bash 支持一个额外的特性，即使用 **^R**(可以认为它是“重新调用”键)。按 **^R** 键并键入一种模式，然后 Bash 将重新调用包含该模式的最近一条命令。例如，调用最近的一条 **ls** 命令，可以按下 **^R** 键然后再键入 **ls**。

如果得到的命令并不是希望的命令，可以再次按 **^R** 键获得包含该模式的下一条最近的命令。在我们的例子中，再次按 **^R** 键将获得另一条 **ls** 命令。

当看到希望的命令后，就可以按 **<Return>** 键运行该命令，或者对命令进行修改后再按 **<Return>** 键。

名称含义

fc

Bourne shell 家族(Bash、Korn shell)使用 **fc** 命令显示及修改历史列表中的命令。**fc** 是一个功能强大的命令，拥有复杂的语法，因此需要一段时间才能掌握。

名称 **fc** 代表“fix command，修复命令”。这是因为，当命令键入错误时，可以使用 **fc** 修改命令，然后再重新执行命令。

13.17 历史列表：设置大小

shell 将历史列表存储在一个文件中。它在注销时可以自动保存这个文件，登录时可以自动恢复这个文件。这非常重要，因为这意味着一个工作会话中的工作记录将保存到另一个工作会话中。

对于 Bourne shell 家族来说，历史文件将被自动保存和恢复。对于 C-Shell 家族来说，除非设置了 shell 变量 **savehist**(参见下面)，否则历史文件不会自动保存。

重要的是要意识到，当查看历史列表时，您查看的命令也许会跨越若干个工作会话。就像日常生活一样，记住的东西太多会产生相反的效果。基于这一原因，shell 允许通过设置一个变量来设置历史列表的大小。

对于 Bourne shell 家族来说, 需要设置 **HISTSIZE** 环境变量。例如, 要指定历史列表存放 50 条(对大多数人来说已经足够)命令, 可以使用:

```
export HISTSIZE=50
```

对于 C-Shell 家族来说, 需要设置 shell 变量 **history**:

```
set history = 50
```

如果希望保留较多的工作记录, 则只需将这个变量设置成一个较大的数字。如果不设置大小, 那么也没有关系, shell 将使用默认值, 这个值也会工作得很好。

正如前面所述, 如果在注销时希望 C-Shell 或 Tcsh 保存历史列表, 则必须设置 shell 变量 **savehist**。和 **history** 一样, 必须为该变量指定希望保存多少条命令。例如, 希望将一个工作会话中的 30 条命令保存到下一个工作会话中, 可以使用:

```
set savehist = 30
```

提示

如果希望设置历史列表的大小, 那么放置该命令的位置就是初始化文件, 这样在每次登录时都可以自动设置这个变量。第 14 章将对此进行讨论。

13.18 历史列表示例: 避免删错文件

第 25 章中将讨论如何使用 **rm**(remove, 删除)命令删除文件。当使用 **rm** 命令时, 可以指定表示一系列文件的模式。例如, 模式 **temp*** 就代表任何以 **temp** 开头, 后面跟零个或多个字符的文件名; 模式 **extra?** 指任何以 **extra** 开头, 后面跟一个字符的文件名。

rm 命令的危险在于一旦将文件删除, 文件就再也不存在了。如果发现删错了文件, 那么即便只是刚刚按下 <Return> 键, 也没有办法来恢复文件了(我们现在先暂停一会, 以便 Macintosh 用户恢复镇静)。^{*}

假设您希望删除一组名为 **temp**、**temp_backup**、**extra1** 和 **extra2** 的文件。您考虑输入下述命令:

```
rm temp* extra?
```

但是, 您忘了还有一个重要的文件 **temp.important**。如果输入上述命令的话, 这个文件也将被删除。

更好的策略就是先按照 **rm** 命令准备使用的模式来使用 **ls**(list file, 列举文件)命令:

```
ls temp* extra?
```

该命令将列举所有匹配这一模式的文件。如果该列表包含有已经忘记的文件, 例如

^{*} 不管您是否相信, Unix 的 **rm** 命令永远删除文件实际上是一件好事。有经验的 Unix 用户极少会偶然丢失文件, 因为它们在操作之前会仔细地考虑。此外, 他们已经学会为自己的行为负责, 因为他们不能依赖于操作系统来弥补他们的过失。您可以想象, 这样的聪明和自信影响生活的各个方面, 这就是为什么 Unix 人士整体比较成功和完美的原因。

(我们再次暂停一会, 使 Macintosh 用户恢复镇静。)

temp.important，那么您就不能再按计划输入 **rm** 命令。但是，如果文件列表正好符合期望，那么就可以继续前进，通过将 **ls** 替换为 **rm** 并重新执行该命令删除所有的文件。对于 Bourne shell，可以使用：

```
fc -s ls=rm
```

对于 C-Shell，可以使用：

```
^ls^rm
```

您或许会问，为什么要重用上一条命令呢？一旦确认这个模式匹配希望删除的文件，为什么不使用相同的模式简单地键入 **rm** 命令呢？

您当然可以这样，但是，重用 **ls** 命令可以确保准确地获得期望的内容。如果重新键入模式，那么不管多么小心，都有可能产生键入错误，最终删错文件。另外，在许多情况下，修改上一条命令要比重新键入一条命令快。

如果喜欢这一思想，那么您还可以通过使用别名使这一过程更加简单。本章后面将对此进行示范。

13.19 在 shell 提示中显示事件编号&工作目录

本章前面讨论了如何在 shell 提示中放置各种不同类型的信息，包括用户标识、shell 名称等。在本节中，将示范两个随时会发生变化的项：事件编号和工作目录名称。将这两个项放在 shell 提示中可以帮助确定自己正在做什么事情。

为了显示事件编号的当前值，需要使用一个特殊码。这个特殊码根据 shell 的不同而不同，因此我在图 13-7 中进行了汇总。下面是 4 种主要 shell 中显示事件编号当前值的例子。在这些例子中，事件编号被放置在方括号*中，这样在显示时比较漂亮：

```
export PS1="bash[\!]\$ "
export PS1="ksh[\!]\$ "
set prompt = "csh[\!]% "
set prompt = "tcsh[%\!]> "
```

shell	特殊码
Bash	\!
Korn Shell	!
C-Shell	!
Tcsh	%!

图 13-7 在 shell 提示中显示历史列表的事件编号

为了在 shell 提示中显示历史列表的事件编号的当前值，需要使用特定 shell 的特殊码。有关示例请参见正文。

* 在 C-Shell 和 Tcsh 中，碰巧一个!字符后面跟一个右方括号会产生问题。因此，在这些例子中，使用了反斜线来引用!字符。这个问题的原因非常难解，因此不用关心它。

例如，假设当前的事件编号是 57，上面定义的 4 个提示将类似于：

```
bash[57]$
ksh[57]$
csh[57]%
tcsh[57]>
```

此外，事件编号还可以与其他特殊码或者变量结合形成一个更精美的提示。例如，下面有几个提示，它们显示 shell 的名称、工作目录(参见第 24 章)和事件编号(有关在 shell 提示中显示工作目录名称的更多信息，请参见图 13-6)。

出于可读性考虑，我在这些例子中插入了空格，将工作目录放在圆括号中，并且将事件编号放在方括号中。

首先是一个 Bash 的提示。我们使用 `\w` 显示工作目录，`\!` 显示事件编号：

```
export PS1="(\w) bash[\!]$ "
```

在 Korn shell 中显示相同的提示需要一点技巧。因为 Korn shell 中没有显示工作目录的特殊码，因此我们使用 `PWD` 变量。但是，因为 `PWD` 变量的值不时发生变化，所以我们必须使用强引用，而不是弱引用(参见本章前面的讨论)。

```
export PS1='($PWD) ksh[\!]$ '
```

另外，我们也可以使用弱引用，只要确保使用一个反斜线来引用 `$` 字符即可：

```
export PS1="(\$PWD) ksh[\!]$ "
```

最后是 Tcsh 中使用的命令。我们使用 `%~` 显示工作目录，`%!` 显示事件编号：

```
set prompt = "(%~) tcsh[%!]> "
```

那么在 C-Shell 中显示相同提示又要使用什么命令呢？从图 13-6 可以得知，在 C-Shell 提示中显示工作目录名称没有简单的方法，因此无法进行示范。但是，有一种比较复杂的方法可以实现这一点，即使用所谓的“别名”。我们将在本章后面讨论这一思想。

13.20 自动补全

shell 使命令输入比较方便的方法之一就是帮助自动补全键入的单词。这一特性称为自动补全。

例如，您正在输入一条命令，而且需要键入一个非常长的文件名，如：

```
harley-1.057-i386.rpm
```

如果没有其他与该名称相似的文件，为什么还必须键入整个名称？应该只键入几个字符，让 shell 为我们完成剩余的工作。

使用自动补全后，shell 会仔细查看所键入的任何内容。在任何时候，都可以按下一个

特殊的键组合，然后 shell 就会尽量地自动补全当前的单词。如果 shell 不能补全单词，那么它将发出嘀嘀声。下面先举例说明这一概念，然后再进一步讨论细节问题。

假设您有 4 个文件：

```
haberdashery
hardcopy
harley-1.057-i386.rpm
hedgehog
```

如果您键入 **harl**，并且按下自动补全的键组合，这里将没有歧义。因为只有一个文件是以 **harl** 开头的，所以 shell 将自动填充该文件名的剩余部分。

但是，如果键入了 **har** 并按下了自动补全键，会发生什么情况呢？因为有两个文件以 **har** 开头，所以 shell 将发出嘀嘀声提示所键入的内容有歧义性。

此时，您有两个选择。一是键入更多的字符并再次尝试自动补全。或者，如果不确定该键入的内容，则可以按第二个自动补全键，使 shell 显示所有可能匹配的文件列表。

对于我们的例子来说，如果键入 **har** 并按第二个键组合，那么 shell 将显示：

```
hardcopy
harley-1.057-386i.rpm
```

然后就可以键入一个 **d** 或者一个 **i**，并告诉 shell 自动完成工作。

正如所见，为了使用基本的自动补全功能，只需记住两个不同的键组合。基于历史原因，在各个 shell 中这些键各不相同，因此图 13-8 中对此进行了汇总。第一个键组合告诉 shell 尝试补全当前单词。第二个键组合告诉 shell 显示与已键入内容匹配的所有可能的自动补全。

shell	补全单词	显示所有可能
Bash	<Tab>	<Tab><Tab>
Korn Shell	<Esc><Esc>	<Esc>=
C-Shell	<Esc>	^D
Tcsh	<Tab>	^D

图 13-8 自动补全键

基本的自动补全特性使用两种不同的键组合。第一种键组合告诉 shell 尝试补全当前正在键入的单词。如果这个键组合不起作用，可以使用第二种键组合，使 shell 显示与当前模式匹配的所有可能的补全。

为了说明自动补全的工作方式，这里首先示范几个例子，大家可以在自己的计算机上试一试。但是，在开始之前，希望您确保自己的 shell 中自动补全功能是打开的。对于 Bash、Korn shell 和 Tcsh 来说，自动补全功能总是打开的。但是，对于 C-Shell 来说，需要设置 **filec** 变量才能打开自动补全功能。实现这一目的的命令为：

```
set filec
```

放置这条命令的最好位置就是初始化文件，这样在每次启动新 shell 时都会自动地设置

这个变量。第 14 章中将对此进行示范。

为了返回到我们的例子中体验自动补全特性，我们需要几个拥有相似名称的文件。我们可以使用 **touch** 命令创建这些文件。这些文件分别是 **xaax**、**xabx**、**xacx** 和 **xccx***。下面是创建这些文件的命令：

```
touch xaax xabx xacx xccx
```

(我们将在第 25 章中讨论 **touch** 命令。现在，您只需知道这是创建空文件的最简单方法即可。)

现在使用 **ls -l** 命令演示自动补全特征，该命令将列举文件名以及其他信息。首先示范当补全文件名时会发生什么事情。键入下述内容，然后停止，但不要按<Return>键：

```
ls -l xc
```

现在，查看图 13-8，选择适合自己 shell 的“补全单词”组合键。也就是说，对于 Bash 或 Tcsh，按<Tab>键；对于 C-Shell，按<Esc>键；对于 Korn shell，按<Esc>键两次。

因为没有歧义性，所以 shell 将补全这个文件名。您将看到：

```
ls -l xccx
```

现在就可以按<Return>键输入该命令了。

这种类型的自动补全称为文件名补全(filename completion)。自动补全还有其他几种类型，我们稍后讨论。

现在，让我们看看当 shell 不能补全文件名时会发生什么事情。键入下述内容，然后停止，但不要按<Return>键：

```
ls -l xa
```

再一次按下自己 shell 的“补全单词”键组合(<Tab>、<Esc>或者<Esc><Esc>)。这一次，shell 将发出嘀嘀声，因为没有单个的文件名匹配键入的内容(实际上，有 3 个文件名与键入的内容匹配)。键入字母 **b**，然后停止，但不要按<Return>键：

```
ls -l xab
```

现在再次按补全键组合。这一次，shell 将能够完成补全，因为只有一个文件名(**xabx**)与 **xab** 匹配。按<Return>键执行这条命令。

最后一个例子。键入下述内容，然后停止，但不要按<Return>键：

```
ls -l xa
```

这一次，查看图 13-8，选择适合自己 shell 的“显示所有可能”组合键。也就是说，对于 Bash，按<Tab>键两次；对于 Korn shell，按<Esc>=键；对于 C-Shell 或 Tcsh，按^D(Ctrl-D)键。这样将告诉 shell 列举所有可能的匹配。

shell 将显示各种可能的匹配，然后再重新显示已经键入的内容，以便您补全命令。您

* 为了防止您觉得奇怪，我根据我的 4 位前女友命名了这些文件。

将看到：

```
xaax xabx xacx
ls -l xa
```

现在您可以选择自己喜欢的方式补全命令，并按<Return>键执行命令。

最后，当结束体验时，为了保持系统整洁，需要移除这 4 个练习文件：

```
rm xaax xabx xacx xccx
```

这是最后一个例子。无论何时，当前工作的目录就称为“工作目录”（参见第 24 章）。有时候，可能希望改变工作目录，为了实现这一目的，需要使用 **cd**(change directory, 改变目录)命令，后面跟一个目录的名称。有时候您会发现需要键入一个很长的目录名，特别是当您是一名系统管理员时。当出现这种情况时，最好使用自动补全特性。

例如，假设您使用的是一个 Linux 系统，您希望改变到下述目录中：

```
/usr/lib/ImageMagick-5.5.7/modules-Q16/filters
```

您可以键入 **cd** 命令，后面跟着这个非常长的目录名。但是，键入最低数量的字符并使用自动补全特性将简便许多。在这个例子中，对于 Bash，可以键入：

```
cd /us<Tab>/li<Tab>/Im<Tab>/mo<Tab>/fi<Tab><Return>
```

如果使用的是不同的 shell，那么自动补全键可能有所不同，但是思想是相同的：shell 可以帮助我们自动完成键入，为什么还要键入一个长名称呢？

13.21 自动补全：高级应用

在第 11 章中，当解释名称 C-Shell 和 Tcsh 的来源时，曾提到原始 Tcsh 的创建者 Ken Greer 在 PDP-10 计算机上工作时，使用的是 TENEX 命令解释器(与 shell 相似)。

TENEX 使用非常长的命令名称，因为这样的命令易于理解，但同时键入完整的命令比较烦人。一个称为“命令补全”的功能被用来完成这些工作。因此，在键入命令时，只需键入少数几个字母，然后按下<Esc>键即可。命令解释器将把已经键入的内容扩展成整个命令。Greer 将这一特性添加到他所编写的新 C-Shell 中，在命名这个 shell 的名称时，他称之为 Tcsh，其中“T”就是指 TENEX。

在上一节中，我们使用自动补全特性帮助键入文件的名称，而这是我们大多数时候使用自动补全特性的方式。但是，由 Tcsh 的故事可以知道，自动补全特性并不是一个新思想。此外，自动补全特性不止用于补全文件名。实际上，现代的 shell 支持自动补全许多不同类型的名称。

各个 shell 之间自动补全特性的细节方面各不相同，而且还相当复杂。实际上，大多数 shell，特别是 Bash 和 Zsh(第 11 章提及)，提供了许多自动补全特性，这些特性您可能三辈子都用不完。大多数时候，只需要使用上一节中讨论的技巧。在本节中，将深入解释一些

技巧。如果希望了解更多细节，可以显示 shell 的说明书页并搜索 “completion(补全)”。

通常，自动补全有 5 种类型。尽管所有现代的 shell 都支持文件名自动补全(这也是最重要的一种自动补全)，但并不是所有的 shell 都支持这 5 种类型。出于参考目的，图 13-9 说明了各种 shell 所支持的自动补全类型。

shell	自动补全	补全对象
B K C T	文件名补全	文件和目录
B . . T	命令补全	命令，包括路径名
B . . T	变量补全	变量
B . C T	用户标识补全	系统上的用户标识
B . . .	主机名补全	局域网上的计算机

图 13-9 自动补全类型

自动补全允许键入名称的一部分，让 shell 补全剩余部分。通常，自动补全有 5 种类型，每种类型补全一种不同类型的名称。详情请参见正文。

最左边的一列说明哪个 shell 支持这种类型的自动补全：B=Bash，K=Korn Shell，C=C-Shell，T=Tcsh。圆点表示相应的 shell 不支持该特性。

我们已经讨论了文件名补全。命令补全(只适用于 Bash、Tcsh)与此相似。当在一行的开头键入命令时，可以使用自动补全帮助键入命令的名称。命令可以是外部命令、内置命令、别名，也可以是函数。

例如，假设您使用的 shell 是 Bash 或者 Tcsh，而您希望输入 **whoami** 命令显示自己的用户标识(参见第 8 章)。您先键入：

whoa

然后按<Tab>键，shell 将自动补全这条命令。

变量补全(Bash、Tcsh)在输入带有\$字符的单词时发挥作用。shell 将假定您准备键入变量的名称。

例如，假设您使用的是 Bash，而您希望显示一个环境变量的值，但是忘记了它的名称。您能够记起的就是这个变量以字母 **H** 开头。因此，可以键入：

echo \$H<Tab><Tab>

对于 Tcsh，可以在**\$H** 之后按[^]**D**(Ctrl-D)键：

echo \$H[^]D

shell 列举所有以字母 **H** 开头的变量。例如：

**HISTCMD HISTFILESIZE HOME HOSTTYPE
HISTFILE HISTSIZE HOSTNAME**

您认出需要的变量是 **HOSTTYPE**，因此继续键入该变量名称的足够部分，从而使该



名称能够被识别, 按<Tab>键(由 Bash)完成作业:

```
echo $HOST<Tab>
```

对于 Tcsh, 可以使用:

```
echo $HOST<Esc><Esc>
```

用户标识补全(也称为用户名补全)在 Bash、C-Shell 和 Tcsh 中可用。除了单词必须以~(波浪号)开头之外, 它与变量补全相似。这是因为语法~userid 是用户标识的 home 目录的缩写。

最后, 主机名补全只在 Bash 中可用, 它将补全局域网中计算机的名称。主机名补全用于以@(at 符号)字符开头的单词。例如, 如果正在键入一个电子邮件地址, 则可以使用主机名补全。

提示

当知道希望键入什么, 但是记不清完整名称时, 自动补全特别有用。

例如, 如果您使用的是 Bash, 而且希望输入一个以 lp 开头的命令, 但是您又记不清是哪条命令了, 那么只需键入 lp<Tab><Tab>即可(对于 Tcsh, 使用 lp^D)。

同理, 通过键入 \$<Tab><Tab>(或者 \$^D)可以列举所有变量的名称。试试看。

13.22 为好玩和赌注使用自动补全

您还记得第 7 章中的程序员和公主的神话故事吗? 在这个故事中, 一位英俊年青的程序员能够通过不必按<Return>键或者^M 而输入一条命令, 从而挽救了漂亮的公主(他使用的是^J)。下面是一些更酷的事情: 如何使用自动补全挣些钱, 同时打动女孩(假设您是男孩)。

下一次出席 Linux 用户聚会时, 看看周围的爱好者, 找一个看上去有钱的人。因为他是一位 Linux 爱好者, 所以您知道他使用的是 Bash。和他打一个小赌(假如说 5 美元), 告诉他您不按<Return>键就可以列出所有环境变量的名称。当他接受这个赌注时, 输入:

```
env^M
```

现在他知道您如何骗他了, 因此再下一个两倍的赌注。这一次, 您答应不使用<Return>键或^M。当他接受这个赌注时, 输入:

```
env^J
```

下面要使用杀手锏了。这一次下一个三倍的赌注, 并且答应不使用<Return>键、^M 或^J。而且, 为了使这个任务看上去更难, 甚至还可以答应不键入命令。

现在, 您将吸引大量其他 Linux 爱好者的注意, 他们也想参加到这个赌局中来。要他们将钱放在桌子上, 一旦收集完所有的赌注, 只需输入:

```
$<Tab><Tab>
```

在您收完桌上的钱向外走时，回过头来看看那些家伙并对他们说：“难道你们没有听说过 RTFM 吗？”

13.23 命令行编辑：bindkey

在前面几节中，讨论了几个相关的主题：在键入命令时进行修改、使用历史列表及使用自动补全。在阅读这几节的内容时，要注意两件事情。首先，3 个较新的 shell，即 Bash、Korn shell 和 Tcsh，相对于比较老的 C-Shell 提供了更丰富的特性。其次，似乎有一个底层的线程将所有这些绑定在一起。

实际上就是这种情况。这里的通用原理就是**命令行编辑**，只有较新的 shell 才支持这一特性，C-Shell 不支持这一特性。命令行编辑是一种强大的工具，允许使用许多不同的命令操纵在命令行上键入的内容，包括使用历史列表和自动补全的能力。

前面多次讲过 Unix 文本编辑器主要有两个：**vi**(发音为“vee-eye”)和 Emacs。无论如何，您必须至少学习使用其中一种编辑器。实际上，本书中有一整章是专门介绍 **vi** 文本编辑器的(第 22 章)。

vi 和 Emacs 都提供有一组数目众多、功能强大的命令，可以用来查看和修改文本文件。这些命令经过精心的设计，适合于在任何场合编辑任何类型的文本。特别地，shell 允许使用 **vi** 命令或者 Emacs 命令(自己选择)查看和修改在命令行上键入的内容以及历史列表。

由于 **vi** 的命令与 Emacs 的命令相差很大，因此 shell 只允许一次使用一种编辑器。默认情况下，shell 假定使用的是 Emacs 命令。我们称之为 **Emacs 模式**。但是，如果希望的话，也可以修改到 **vi** 编辑器。如果修改到 **vi** 编辑器，则称 shell 处于 **vi 模式**中。

从一种命令编辑模式改变到另一种命令编辑模式的方法取决于使用的 shell。对于 Bash 和 Korn shell 来说，需要设置一个 shell 选项，即 **emacs** 或者是 **vi**(shell 选项在第 12 章中已解释过)。因此，需要使用下述两条命令中的一条：

```
set -o emacs
set -o vi
```

对于 Tcsh 来说，需要使用 **bindkey** 命令。可以附带 **-e**(Emacs)或者 **-v**(vi)选项：

```
bindkey -e
bindkey -v
```

这两条命令最好放在初始化文件中，从而在每次登录时都可以自动执行它们。第 14 章将示范如何设置。

当编辑常规的文本文件时，对于初学者来说，**vi** 是最好的选择。这是因为，尽管 **vi** 比较难学，但是 Emacs 更加难学。因此，如果您是一名初学者，当学习如何编辑文件时，我推荐先学习 **vi**，而不是 Emacs。

但是，当编辑历史列表和命令行时，Emacs 实际上要比 **vi** 更容易使用。原因在于，大多数时候，只需要在历史列表中向上和向下移动，或者对命令行进行微小的修改。对于

Emacs 来说, 这比较直接。**vi** 编辑器更加复杂, 因为它有两种不同的模式: 命令模式和插入模式。在使用 **vi** 之前, 需要学习如何在两种模式之间来回切换(第 22 章中将详细讨论)。基于这一原因, 所有的 shell 默认都使用 Emacs。

当在本章前面讲授如何使用<Up>和<Down>在历史列表中移动, 以及如何对命令行进行基本的修改时, 我实际上示范的是简单的 Emacs 命令。因此, 尽管那个时候您还没有意识到, 但是您已经在命令行编辑中使用了 Emacs。实际上, 如果 shell 处于 **vi** 模式中, 您将发现光标移动键与期望的工作方式不同。

vi 和 Emacs 都提供有大量操纵历史列表和命令行的方法。但是, 除非您学会如何使用一种编辑器, 否则所有方法对您都没有帮助。基于这一原因, 我不准备解释高级命令行编辑的细节。如果您想学, 那么在学会了如何使用编辑器之后, 您可以去体验 **vi** 或者 Emacs 命令。那时候, 再回到本章, 阅读本节的剩余部分(对于 **vi** 来说, 要等到阅读完第 22 章内容之后)。

为了自学命令行编辑, 首先使用 **set** 或者 **bindkey** 命令将 shell 设置成 **vi** 模式或者 Emacs 模式。现在, 假设您使用一个包含历史列表的不可见文件。在任何时候, 您都可以从这个文件中复制一行到命令行, 然后在命令行上对文本进行修改。无论何时, 当按下<Return>键运行命令时, 命令行的内容就添加到不可见文件(也就是历史列表)的末尾。

在脑海里记住这种情形, 它的体验相当简单。所需做的工作就是以一种合理的方式使用 **vi** 命令或者 Emacs 命令。首先练习基本的操作: 在不可见文件中来回移动、搜索模式、进行替换等。您将会发现, 一旦习惯了 **vi** 或者 Emacs, 命令行编辑其实相当直接和直观。

如果需要这方面的参考信息, 可以查看特定 shell 的说明书页, 搜索有关命令行编辑的信息。您可能会发现指示有点混乱, 不过一定要有耐心, 因为适应这方面的知识需要一定的时间。

13.24 别名: alias、unalias

别名就是赋予一条命令或者一系列命令的名称。可以将别名作为缩写, 或者使用别名创建已有命令的自定义变体。例如, 假设您发现自己经常输入下述命令:

```
ls -l temp*
```

如果给这条命令赋一个别名 **lt**, 那么就可以通过键入下述命令来简化上一条命令:

```
lt
```

创建别名时需要使用 **alias** 命令。该命令的语法与使用的 shell 有些相关。对于 Bourne shell 家族(Bash、Korn shell)来说, 语法为:

```
alias [name=commands]
```

确定等号的两边不要有空格(创建变量时也有这样的要求)。

对于 C-Shell 家族(C-Shell、Tcsh)来说, 语法几乎相同。唯一的区别就是没有等号:

```
alias [name commands]
```

在两种情况中, *name* 是希望创建的别名名称, 而 *commands* 是一个包含一条或多条命令的列表。

作为示例, 下面我们创建前面提及的别名。第一条命令针对 Bourne shell 家族, 第二条命令(没有等号)针对 C-Shell 家族:

```
alias lt='ls -l temp*'
alias lt 'ls -l temp'
```

注意, 上述例子将命令引用在单引号中。这是因为该命令包含有多个空格和一个元字符(*)。通常, 强引用(单引号)要比弱引用(双引号)好, 因为它们保留元字符的含义, 直至别名执行。

下面再举一个例子, 这个例子为含有两条命令的一串命令创建别名。同样, 第一条命令针对 Bourne shell, 第二条命令针对 C-Shell:

```
alias info='date; who'
alias info 'date; who'
```

一旦创建了这个别名, 就可以在任何时候输入 **info** 来运行这两条命令。

下面是我最喜欢的别名, 它为 **alias** 命令本身创建一个缩写:

```
alias a=alias
alias a alias
```

一旦创建了这个别名, 就可以使用 **a** 代替整个单词 **alias** 来创建别名。例如, 一旦定义了这个别名, 就可以输入:

```
a info='date; who'
a info 'date; who'
```

如果希望修改别名的含义, 则只需重定义这个别名。例如, 如果 **info** 是一个别名, 那么您可以在任何时候使用另一个 **alias** 命令修改它:

```
alias info='date; uname; who'
alias info 'date; uname; who'
```

输入 **alias** 命令和别名的名称就可以查看这个别名的当前值。例如, 要显示别名 **info** 的含义, 可以使用:

```
alias info
```

要同时显示所有的别名, 可以输入没有参数的 **alias** 命令:

```
alias
```

使用 **unalias** 命令可以移除别名。该命令的语法为:

```
unalias name
```

其中 *name* 是别名的名称。例如，要移除刚才定义的别名，可以使用：

```
unalias info
```

如果希望同时移除所有的别名(假如说您只是想试验一下)，可以使用 **unalias** 命令加 **-a** 选项(针对 Bourne shell)或者 ***** 字符(针对 C-Shell)：

```
unalias -a
unalias *
```

您还记得 **type** 命令吗？我们在本章前面讨论过(指定一条命令的名称，然后 **type** 就可以告诉您命令的类型)。使用 **type** 命令可以查看某条特定的命令是不是别名。例如，假设您定义了如上所示的别名 **info**，然后输入：

```
type info
```

您将看到一个与下述内容相似的消息：

```
info is aliased to 'date; who'
```

可以想象，您可能希望开发一整套经常使用的别名。但是，每次登录时都重新键入别名命令非常烦人。实际上，您可以将所有自己喜爱的别名定义放置在一个初始化文件中。每当启动新 shell 时，这些别名命令就会自动执行。第 14 章将示范详细过程。

13.25 临时挂起别名

别名的一个非常常见的应用就是让用户在每次运行特定命令时，能够方便地使用相同的选项。

例如，**ls** 命令(第 24 章中讨论)列举一个目录的内容。当使用 **ls** 命令本身时，获得的是一个“短”列表；当 **ls** 命令使用 **-l** 选项时，获得的是一个“长”列表。

假设您发现，在使用 **ls** 命令时，几乎总是使用 **-l** 选项。那么，为了节省每次键入选项的时间，可以定义下述别名：

```
alias ls="ls -l"
alias ls "ls -l"
```

(第一个定义针对 Bourne shell 家族，第二个定义针对 C-Shell 家族。)

现在，只需简单地输入命令本身就可以显示“长”列表，不再需要键入选项：

```
ls
```

这将生成一个长列表，就像输入下述命令一样：

```
ls -l
```

当使用这样的别名时，您将发现，有时候您希望运行原始的命令，而不是别名。例如，

您希望运行没有 **-l** 选项的 **ls** 命令。

为了临时挂起一个别名(只针对一条命令)，只需在命令名称的开头键入一个 ****(反斜线)字符：

```
\ls
```

这样就告诉 **shell** 运行实际命令本身，而不是别名。在上面的例子中，**shell** 将忽略 **ls** 别名，所以获得的是短列表(默认情况)。

13.26 别名示例：避免删错文件

在本节中，我将示范如何将别名与从历史列表中调取的命令结合在一起，生成一个特别方便的工具。

在本章前面，我们讨论了一个例子。在这个例子中，我们打算使用 **rm**(remove, 删除)命令删除所有匹配特定模式的文件。前面讨论的例子为：

```
rm temp* extra?
```

为了确保在删除文件的过程中不犯错误，在执行实际删除之前，我们应该先检查即将使用的模式。也就是说，对 **ls** 命令使用相同的模式：

```
ls temp* extra?
```

如果 **ls** 命令所列举的文件是我们需要的，就可以进行删除。否则，我们需要再尝试一种不同的模式，直至获得所希望的文件列表。通过这种方式，我们确保 **rm** 命令可以准确地删除希望删除的文件。这一点非常重要，因为一旦 **Unix** 将文件删除，这个文件就永远无法恢复。

那么，假设 **ls** 命令查找到的文件就是我们希望删除的文件。这样，我们就可以使用相同的模式输入 **rm** 命令。但是，如果在输入 **rm** 命令的过程中有键入错误呢？我们最终可能会删错文件。更好的方法就是让 **shell** 为我们完成这个工作。为此，需要从历史列表中调取 **ls** 命令，将 **ls** 修改为 **rm**，然后再执行修改后的命令。

对于 Bourne shell 家族(Bash、Korn shell)的成员，需要使用：

```
fc -s ls=rm
```

为了使这条命令更容易使用，我们定义一个命名为 **del** 的别名：

```
alias del='fc -s ls=rm'
```

对于 C-Shell 家族(C-Shell、Tcsh)的成员，通常使用：

```
^ls^rm
```

但是，由于技术上的原因，这里无法再深入，这条命令不能应用别名。相反，我们需要使用下述命令：

```
rm !ls:*
```

很明显，这里存在一定的技巧(Unix 中到处是需要技巧的命令)。非正式地讲，我们请求 shell 从最近一条 **ls** 命令中提取参数，并使用它们运行一条 **rm** 命令。最终效果是使用和 **ls** 命令相同的参数运行 **rm** 命令。

为了使这条命令更容易使用，我们可以定义一个别名。注意我们引用了 **!**，从而保留它的含义(您能理解该含义吗?)

```
alias del 'rm \!ls:*
```

一旦定义了这个 **del** 别名，就可以使用下述过程删除匹配特定模式的文件。这里的出色之处在于，不管使用哪一种 shell，该过程都是相同的。

首先，我们输入 **ls** 命令，以及描述希望删除的文件的模式。例如：

```
ls temp* extra?
```

如果该模式显示的文件名称符合我们的期望，接下来可以输入：

```
del
```

这样就可以将文件删除。

如果模式无法满足期望，则需要重新输入 **ls** 命令和一个不同的模式，直至获得期望的结果，然后再使用 **del**。通过这种方式，就不会再因为使用错误的匹配模式而删错文件了。

如果您养成了在 **del** 命令之前使用 **ls** 的习惯，那么我敢保证，总有一天，您将挽救自己于一场大灾难中。实际上，我有这个命题的数学证明——使用数学归纳和超几何函数得出的，这是一种技巧，光这个技巧就可以值回买书的钱了(遗憾的是，该证明的解释已经超出了本书的讨论范围)。

13.27 别名示例：从历史列表中重用命令

在本章前面，已经解释了 Bourne shell 家族和 C-Shell 家族使用不同的命令访问历史列表。具体而言，Bourne shell 家族(Bash、Korn shell)使用 **fc** 命令，而 C-Shell 家族(C-Shell、Tcsh)使用 **history** 和 **!** 命令。

最初的历史列表功能是为 C-Shell 编写的。那个时候，这是一个突破。实际上，现在它仍然有用并且容易使用。后来，Korn shell 开发出了一个功能更加强大的系统，该系统使用 **fc** 命令。然而，**fc** 命令的语法设计得不好，而且命令本身的细节也难以记住和使用。但是，通过使用别名，我们可以使 **fc** 命令看上去就像 C-Shell 系统一样。

首先，我们定义一个名为 **history** 的别名，该别名使用带 **-l**(list, 列举)选项的 **fc** 命令显示历史列表中的命令行：

```
alias history="fc -l"
```

为了使这个命令更简单，可以将 **history** 缩写为 **h**：

```
alias h=history
```

这是我最喜爱的别名之一，我在每个 shell 中都使用它，即便是 C-Shell 和 Tcsh。毕竟，有谁希望一遍又一遍地键入单词 **history** 呢？*

接下来，我们定义别名 **r**(recall, 调取)来取代 **fc -s**，即从历史列表中调取命令并重新执行命令的命令：

```
alias r="fc -s"
```

现在，无论何时，我们都可以方便地执行之前输入的最后一命令。只需使用 **r** 别名即可：

```
r
```

如果希望对命令进行修改，则只需指定一个旧模式和一个新模式。例如，假定我们刚刚键入了下面一条命令：

```
vi tempfile
```

该命令将启动 **vi**，编辑一个名为 **tempfile** 的文件。我们决定再次运行这条命令，编辑一个名为 **data** 的文件。我们所需做的全部事情就是键入：

```
r tempfile=data
```

使用历史列表中一个具体的行也相当容易，只需指定事件编号(行号)即可。例如，假设历史列表如下所示：

```
20 cp tempfile backup
21 diff backup backup1
22 whoami
23 date
24 vi tempfile
25 vi data
```

您现在想知道时间，因此希望重新执行 **date** 命令，事件编号为 23：

```
r 23
```

接下来，您希望重新执行命令 20。此外，您还希望把 **tempfile** 文件修改为 **data** 文件：

```
r 20 tempfile=data
```

如果指定一个或多个字符，那么 shell 将重新执行以这些字符开头的最近一条命令。例如，为了重新执行以 **di** 开头的最近一条命令(在这个例子中，是编号为 21 的 **diff** 命令)，可以使用：

* 基于相同理由，又有谁愿意一遍又一遍地输入单词 **alias** 呢？这就是为什么我建议要给 **alias** 命令创建一个别名的原因：

```
alias a=alias
```

```
r di
```

如果希望重新执行 **date** 命令, 则可以指定以 **d** 开头的最近一条命令:

```
r d
```

经过小小的练习之后, 这样的替换可以节省您大量的时间和精力。

在结束本节内容之前, 先给出特定 shell 中使用 **history**、**h** 和 **r** 别名的一些具体建议。

Bash: 正如本章前面解释的, Bash 既提供有 **fc** 命令, 也提供有 **history** 和 **!** 命令。但是, 您自己需要创建 **h** 和 **r** 别名:

```
alias h=history
alias r="fc -s"
```

Korn shell: Korn shell 使用 **fc**, 而且还提供有 **history** 命令和已经定义好的别名 **r**, 因此不再需要定义它们。但是, 为了方便, 还需要添加 **h** 别名:

```
alias h=history
```

C-Shell 和 Tcsh: 正如本章前面解释的, 这两种 shell 都提供有 **history** 命令, 以及一种修改及重用历史列表中命令的简单方法。为了方便, 还需要添加 **h** 别名:

```
alias h history
```

这些别名有两种好处: 首先, 它们使历史列表的使用更为简单; 其次, 无论使用哪一种 shell, 它们都允许以相同的方式访问历史列表。

13.28 别名示例: 在 shell 提示中显示工作目录名称

本节的目的是解决只在 C-Shell 中存在的一个具体问题。但是, 不管您使用哪一种 shell, 由于我们也将讨论几个适合于所有 shell 的重要概念, 因此我希望您仔细阅读本节的内容, 思考讨论中出现的各种理念。

在本章前面, 讨论了如何在 shell 提示中显示工作目录的名称。在那个讨论的末尾, 我提到 C-Shell 没有一种实现这一目标的简单方法。但是, 有一种使用别名的复杂方法可以实现这一目标, 而我们准备在这一节讨论这一方法。

该讨论中将涉及到目录, 而目录将在第 24 章中讨论。现在, 您只需知道目录中存放着文件, 而“工作目录”就是当前工作所在的目录。您可以随时从一个工作目录改变到另一个工作目录, 在这样做时, 能够从 shell 提示中看出新目录的名称将带来极大的便利, 因为这样可以随时明了自己当前位于哪里。

以这种方式在 Bash(使用 **\w**)、Korn shell(使用 **\$PWD**)和 Tcsh(使用 **%~**)中显示工作目录的名称比较方便。下面是一些完成该作业的命令示例。出于可读性考虑, 这些命令在圆括号中显示工作目录的名称:

```
export PS1="( \w) bash$ "
```

```
export PS1='($PWD) ksh$ '
set prompt = "(%~) tcsh> "
```

这些命令可行的原因在于当改变工作目录时，shell 会自动地更新 shell 提示中的代码或者变量。

可以确定的是，C-Shell 中有一个 **PWD** 变量。但是，如果将这个变量放在 shell 提示中，您将会发现这个变量不会自动地更新。这是因为 C-Shell 比其他 3 种 shell 都要古老，它没有提供这种能力。

解决这一问题的方法就是使用一个别名，在每次改变工作目录时，这个别名都重新定义 shell 提示。首先，需要回答一个问题：我们使用哪条命令来修改工作目录呢？答案是 **cd**(change directory, 改变目录)命令。

我们将在第 24 章中详细地讨论 **cd** 命令。现在，我告诉您，为了改变到一个新的目录，只需键入 **cd** 命令，后面跟着这个新目录的名称即可。例如，如果希望改变到 **bin** 目录，可以输入：

```
cd bin
```

第 12 章中讲过，在任何时间，C-Shell 都在两个不同的变量中维护工作目录的名称，这两个变量是 **cwd**(shell 变量)和 **PWD**(环境变量)。无论何时，当使用 **cd** 命令改变工作目录时，这两个变量都被更新。

因此，我们的计划就是创建一个别名，重新定义 **cd**，从而使它完成两件事情：(1)根据指定的内容改变工作目录；(2)使用 **cwd** 或者 **PWD** 变量重新定义 shell 提示，以反映新的工作目录。完成这一任务的别名如下：

```
alias cd 'cd \!* && set prompt="($PWD)% "'
```

为了理解这个别名的工作方式，您需要明白 **&&** 将两条命令分隔开。**&&** 的含义就是首先运行第一条命令，然后根据第一条命令是否正常结束来运行第二条命令。如果第一条命令失败，那么第二条命令就不会执行。换句话说，如果由于某些原因，**cd** 命令失败了，那么 shell 提示的更新就不会进行。

别名 **cd** 首先执行下述命令：

```
cd \!*
```

表示法 **\!*** 指在原始命令行上键入的任何参数。通过这种方式，命令行上的原始参数就传递给别名中的 **cd** 命令(这与编程相关，因此如果您无法理解也不用担心)。

如果第一条命令正常结束，那么 **PWD** 变量将被 shell 自动更新。然后，我们可以运行第二条命令：

```
set prompt="($PWD)% "
```

这条命令修改 shell 提示，在圆括号中显示工作目录的名称，后面跟一个 **%** 字符，再跟一个空格。这个别名的工作方式就是这样的。

整条命令可行的原因在于别名扩展在 shell 解析和解释命令行之前完成。例如，假设我

们定义了上述别名 `cd`，并输入：

```
cd bin
```

那么 shell 所做的第一件事情就是扩展别名。从内部而言，这个命令行已经修改成：

```
cd bin && set prompt="($PWD)% "
```

然后 shell 执行 `cd` 命令，接着再执行 `set` 命令。

一旦定义好基本的别名，就可以定义更复杂的别名。例如，为什么 shell 提示只显示工作目录和 `%` 字符呢？就不能显示更多其他内容吗？

下述别名就定义了一个更复杂的提示，在这个提示中，我们在圆括号中显示工作目录，后跟一个空格和 shell 的名称，接下来在方括号中显示事件编号，并显示一个 `%` 字符和一个空格：

```
alias cd 'cd \!* && set prompt = "($PWD) csh[\\!\\!]% "'
```

通过这种方式定义的典型 shell 提示如下所示：

```
(/export/home/harley) csh[57]%
```

这就是在初始化文件中放置的别名类型，从而使提示可以自动更新。初始化文件将在第 14 章中讨论。

最后补充一点。上一个例子中的 `!` 字符被引用了两次(通过两个反斜线)。第一个反斜线在 `!` 第一次被解析时引用 `!`，作为 `alias` 命令的一部分。第二个反斜线在 `!` 第二次被解析时引用 `!`，作为 `set` 命令的一部分。

这就是我希望大家一定要理解的概念：当某些内容需要被解析不止一次时，该内容必须被引用多次。请花点时间考虑这一概念，确保理解这一概念。

13.29 练习

1. 复习题

1. 什么是字符数字式字符？什么是元字符？列举 3 种元字符并解释它们的用途。
2. 在 Unix 世界中，一些字符拥有绰号。例如，撇号通常指“引号”或者“单引号”。那么 Unix 人士为下述字符指定了什么绰号：星号、输入/回车、感叹号、点号、引号以及竖线？
3. 引用字符的 3 种不同方式各是什么，它们之间有什么区别？
4. 什么是内置命令？在什么地方查找内置命令的文档资料？
5. 什么是搜索路径？如何显示搜索路径？
6. 什么是历史列表？最简单、最常见的历史列表应用就是重新执行前面的命令。这是如何完成的呢？如果使用 Bash 或者 Tcsh，那么如何调取、编辑然后再执行前面的命令呢？

7. 什么是自动补全？自动补全有多少种类型？简述各种类型的自动补全各完成什么补全。哪种类型的自动补全是最重要的？

2. 应用题

1. 如何修改 Bash 的 shell 提示，以显示用户标识、工作目录和当前时间？在 Tcsh 的 shell 提示中又是如何实现相同修改的呢？

2. 什么是命令替换？使用命令替换创建一条命令，该命令显示 “These userids are logged in right now:”，后面是用户标识列表。

3. 输入下述命令：

```
echo "I am a very smary person."
```

使用历史列表工具，将该命令中的 “smary” 修改为正确的拼写 “smart”。

4. 您的工作目录中包含下述文件(只包含这些文件)：

```
datanew dataold important phonenumbers platypus
```

使用自动补全时，引用各个文件需要键入的最低字符数各是多少？

3. 思考题

1. 在本章中，我们讨论了几种帮助快速输入命令的工具：历史列表、自动补全和别名。这些工具都比较复杂，需要花一些时间掌握。一些人不愿意花时间学习这些工具，因为对他们来说，这并不值得学习。其他人则爽快地接受这些工具。那么，什么类型的人对尽可能快地输入命令有强烈的需要呢？

2. 创建许多专门的别名有什么优点？又有什么缺点？



使用 shell：初始化文件

本书共有 4 章讨论 shell，这是最后一章。第 11 章从总体上讨论了 shell，第 12 章和第 13 章讨论了熟练使用 shell 所需的基本概念。

在本章中，将讨论最后一个主要主题：初始化文件。您将会喜欢上这一章，因为在您阅读这一章时，前面所学的内容将被组合在一起。在这一过程中，您将逐渐欣赏到 shell 之美。

14.1 初始化文件和注销文件

设计 Unix shell 的程序员是那些知道用户自定义工作环境的价值的程序员。根据这一目标，所有的 shell 都允许指定某些命令按照自己的需要自动执行。您的工作就是使用这些命令准确地按照自己的希望设置工作环境。下面是具体过程。

首先，您要创建两个特殊的文件，即初始化文件(initialization file)。在第一个文件，即登录文件(login file)中，存放着所有希望在每次登录时自动执行的命令。在第二个文件，即环境文件(environment file)中，存放着所有希望在新 shell 启动时自动执行的命令。

例如，您可能希望在每次登录时设置一些具体的变量。如果使用的是多用户系统，那么您还有可能运行 **users** 命令显示还有谁登录系统。此外，您可能使用环境文件为每次启动的新 shell 设置特定的 shell 选项并定义特定的别名。

一旦您创建了初始化文件，shell 就会在合适的时间查找它们并自动运行命令。如果需要修改，则只需修改这些文件即可。

为了提供更多的定制功能，一些 shell 还支持注销文件(logout file)。注销文件中存放注销时自动运行的命令。例如，您可能希望在注销时运行 **date** 命令，以显示当前的时间和日期。

提示

以前，有一个很酷的程序叫 **fortune**。每次运行这个程序时，它都显示一个笑话或者一条名言，这些笑话和名言都是从一大堆有趣的娱乐材料中随机选取的。许多人在自己的注销文件中放置了一条 **fortune** 命令，从而在每次注销时都可以看到一些有趣的东西。

遗憾的是，大多数现代的 Unix/Linux 系统中并不包含 **fortune** 命令。但是，网络上已经有了这个程序，您可以下载并安装这个程序。如果安装了这个程序，您将会发现 **fortune** 是一个放置在注销文件中的极好的命令。

总而言之，登录文件、环境文件和注销文件允许您在 3 个不同的时间执行那些希望自动执行的命令，这 3 个时间分别为登录时、新 shell 启动时和注销时。

您能明白这种设计的优美之处吗？能够在这 3 个具体的时间运行命令，意味着您拥有准确地按照自己的希望设置工作环境的控制权力。如果您是一名初学者，那么这些文件的功能可能还不明显。但是，一旦您使用 Unix 已达一两年，而且擅长设置变量、创建 shell 脚本，以及定义别名和函数，那么您将感受到初始化文件对整个 Unix 体验的重要性(我们已经在第 13 章中讨论了别名，函数超出了本书的讨论范围)。

这 3 个文件的名称在各个 shell 中各不相同。例如，对于 C-Shell 而言，登录文件称为 **.login**，环境文件是 **.cshrc**，而终止文件是 **.logout**。图 14-1 示范了各个 shell 使用的标准文件名称。仔细地看一看，您就会发现 Bourne shell 家族(Bash、Korn shell)和 C-Shell 家族(C-Shell、Tcsh)使用不同模式的文件名。

shell	登录文件	环境文件	注销文件
C-Shell	.login	.cshrc	.logout
Tcsh	.login	.tcshrc 、 .cshrc	.logout
Bourne Shell	.profile	—	—
Korn Shell	.profile	\$ENV	—
Bash(默认)	.bash_profile 、 .bash_login	.bashrc	.bash_logout
Bash(POSIX)	.profile	\$ENV	.bash_logout

图 14-1 初始化文件和注销文件的名称

Unix shell 允许自定义工作环境，这通过在特定时间自动运行的特殊文件中放置命令来实现。初始化文件有两种：登录文件(在登录时运行)和环境文件(在新 shell 启动时运行)。一些 shell 还允许使用注销文件(在注销时运行)。从这个表中可以看出，这些文件在各个 shell 之间不尽相同。最初的 Bourne shell 只使用一个称为 **.profile** 的登录文件。

注意：(1)Korn shell 和 Bash(POSIX 模式)不使用标准的环境文件名称。此时，可以将 **ENV** 变量设置为希望使用的任意文件的名称。(2)如果 shell 不支持注销文件，可以通过捕获 **EXIT** 信号使用注销文件(详情请参见正文)。

注意，所有的初始化文件和注销文件都采用一个以 **.**(点)字符开头的文件名称。这样的文件称为“点文件(dotfile)”，我们将在本章后面讨论它们，并在第 24 章中进一步展开讨论。现在，您只需知道 3 件事情。首先，点(也就是一个句点)在文件名中是一个合法的字符。其次，名称开头的点拥有特殊的含义。第三，当谈论点文件时，应将其发音为“dot”。例如，当谈论 **.login** 文件时，要说“dot-login”；当谈论 **.profile** 文件时，要说“dot-profile”。

14.2 初始化文件和注销文件的名称

C-Shell 家族中初始化文件和注销文件的名称比较直接和简单。登录文件是 **.login**，环

境文件是`.cshrc`(C-Shell)和`.tcshrc`(Tcsh),而注销文件是`.logout`。出于向后兼容性考虑,如果 Tcsh 找不到名为`.tcshrc`的文件,那么它将查找一个名为`.cshrc`的文件,这样才合理。

Bourne shell 家族使用的名称需要花点时间解释。首先,我们需要回忆第 11 章中的一个重要思想。在 20 世纪 90 年代初,创建了一组称为 POSIX 1003.2 的规范,以描述“标准的” Unix shell。POSIX 标准的大部分内容都根据 Bourne shell 家族模型来确定。实际上,现代的 Bourne shell(Bash、Korn shell、FreeBSD)都遵循 1003.2 标准。

POSIX 标准要求 shell 应该支持登录文件和环境文件,但是没有必要支持注销文件。登录文件的名称应该是`.profile`。但是,为了保持灵活性,环境文件的名称没有固定,而是采取在名为 `ENV` 的环境变量中存放环境文件的名称的方法。例如,如果您是一名 Korn shell 用户,那么您可能将 `ENV` 的值设置为`.kshrc`(我们在后面讨论文件名)。

如果查看图 14-1,将会发现 Korn shell 遵循 POSIX 规范。其登录文件的名称是`.profile`,而环境文件的名称存放在 `ENV` 变量中。

Bash 有所不同,因为它是由特别聪明的程序员(参见第 11 章)创建的,这些程序员将它设计成以两种不同模式运行:默认模式(追求功能和灵活性*)和 POSIX 模式(追求兼容性)。在默认模式中,Bash 支持 POSIX 标准的增强版;在 POSIX 模式中,Bash 严格遵循 1003.2 标准。

通常,Bash 的默认模式已经很好,而这是大多数人最经常使用的模式。但是,如果需要与 POSIX 兼容的 shell,例如说要运行特殊的 shell 脚本,那么您可以随时在 POSIX 模式中运行 Bash**。

在默认模式——您和我通常使用的模式中,Bash 查找一个名为`.bash_profile`或者`.bash_login`的登录文件(使用哪一个文件取决于自己的希望)和一个名为`.bashrc`的环境文件。

在 POSIX 模式中,Bash 遵循和 Korn shell 相同的规则。登录文件命名为`.profile`,而环境文件的名称存储在 `ENV` 变量中。

在两种模式——默认模式和 POSIX 模式——中,Bash 都使用一个名为`.bash_logout`的注销文件。

现在您应该能够理解图 14-1 中的内容了,这意味着我们现在应该将注意力转向一个十分重要的问题:什么类型的命令应该放在初始化文件和注销文件中呢?

在回答这个问题之前,我希望先解释两个与主题无关的问题,一个问题与点文件和 `rc` 文件有关,另一个问题讨论为了创建和编辑文件需要知道些什么内容。

14.3 点文件和 rc 文件

在查看图 14-1 中的文件名时,您是否注意到两件奇怪的事情?首先,所有的文件名都

* 在所有主要 shell 中,比 Bash 提供更多灵活性的唯一一种 shell 是 Zsh(参见第 11 章)。

** 以 POSIX 模式运行 Bash 有两种方法。第一种方法是使用 `--posix` 选项启动 Bash。这一技巧适用于所有的系统:

```
bash --posix
```

第二种方法比较简单,但是只适用于一些系统。

一些 Unix 系统被设置成 `bash` 命令和 `sh` 命令都可以启动 Bash(Linux 中通常就是这种情况)。在这样的系统上,`bash` 命令以默认模式启动 shell,而 `sh` 命令以 POSIX 模式启动 shell。

以一个点号开头；其次，环境文件的名称都以 **rc** 结尾。

以一个点号开头的文件称为**点文件**或者**隐藏文件**。我们将在第 24 章中对此进行讨论，但是，为了帮助理解，这里先作一个概要说明。

大多数时候，基于某些原因，用户希望忽略许多文件。通常，这些文件就是被程序默认使用的配置文件。这种情况的一个极好的例子就是刚刚讨论过的 **shell** 初始化文件。

正如第 24 章中讨论的，列举文件的命令是 **ls**。为了方便起见，**ls** 命令不列举任何以点号开头的文件，除非使用了 **-a**(all files, 所有文件)选项。因此，当以普通方式使用 **ls** 命令时，看不到任何点文件的名称。

这就是为什么所有的初始化文件和注销文件的名称都是点号开头的原因。一旦以自己需要的方式设置好这些文件，那么除非希望修改这些文件，否则就没有理由再考虑它们了。特别是每次列举文件时，都没有必要查看它们。

如果希望列举所有的文件，包括点文件，可以使用 **ls -a** 命令。为了查看该命令的工作方式，请试一试下述两条命令：

```
ls
ls -a
```

接下来，您应该注意到环境文件的名称都以字符串 **rc** 结尾，例如：**.bashrc**、**.cshrc** 和 **.tcshrc**。这是 Unix 程序命名初始化文件的一种常见约定。例如，**vi** 和 **ex** 编辑器(它们相关)使用一个命名为 **.exrc** 的初始化文件；而通用的 Unix 电子邮件程序 **mail** 使用一个命名为 **.mailrc** 的初始化文件。

在使用 Unix 多年之后，您将会遇到许多 **rc** 文件。通常，这样的文件用于存放初始化命令，而且它们几乎总是点文件，从而可以在 **ls** 命令中隐藏。

名称含义

rc 文件

许多 Unix 程序使用名称以 **rc** 结尾的配置文件，例如 **.bashrc**、**.cshrc** 和 **.tcshrc**。标识 **rc** 代表“run commands, 运行命令”，也就是特定程序每次启动时自动运行的命令。

这一名称派生于 CTSS 操作系统(Compatible Time Sharing System, 兼容分时系统)，该系统于 1963 年在麻省理工学院开发。CTSS 中拥有一个称为“runcom”的功能，该功能可以执行存储在一个文件中的一串命令。一些初期的 Unix 程序员使用过 CTSS，因此，当他们创建配置文件时，选择让名称以 **rc** 结尾。

这是 Unix 程序员使用以 **rc** 结尾的点文件命名初始化文件这项长期传统的起点。例如，如果您编写了一个叫 **foo** 的程序，那么您应该将该程序的初始化文件命名为 **.foorc**。这就是为什么在本章前面我建议，如果您是一名 Korn shell 用户，就应该将环境文件命名为 **.kshrc** 的原因。

在谈论这样的文件时，应该将 **rc** 按两个单独的字母发音。例如，**.foorc** 就发音为“dot-foo-R-C”。出于参考目的，图 14-2 示范了最常见的环境文件的发音。当与其他 Unix 人士讨论时，知道这些名称的发音非常重要。

环境文件	发音
<code>.cshrc</code>	“dot-C-shell-R-C”
<code>.tcshrc</code>	“dot-T-C-shell-R-C”
<code>.bashrc</code>	“dot-bash-R-C”

图 14-2 rc 文件名称的发音

许多 Unix 程序使用的初始化文件的名称以一个点号开头，并且以字母 **rc** 结尾。点号可以防止在列举文件时显示这些文件的名称；**rc** 是“run commands(运行命令)”的缩写(完整的解释请参见正文)。本表列举了最常见 shell 环境文件的名称，以及每个名称的最常见发音。您能明白其中的模式吗？

14.4 使用简单的文本编辑器

为了成为一名技能丰富的 Unix 用户，必须能够快速地编辑(修改)文本。如果您是一名程序员，那么情况尤为真切。具体而言，您的登录文件、环境文件和注销文件中都包含有文本，而为了创建或者修改这些文件，您需要使用一个文本编辑器程序。

Unix 中两个主要的文本编辑器是 **vi** 和 Emacs，您需要掌握其中的一个。但是，**vi** 和 Emacs 都比较复杂。实际上，在本书中，有一整章是专门介绍 **vi** 的(第 22 章)。阅读这一章内容，或者自学 Emacs，都需要花不少的时间。因此我提供一些备选方案，如果您使用的是桌面环境(例如 Gnome 或 KDE)(参见第 5 章)，那么在桌面环境中就有简单的基于 GUI 的编辑器，您可以使用它们创建和修改小的文本文件。如果您使用的是 CLI(命令行界面)，那么或许有基于文本的简单编辑器可供使用，直至您学习 **vi** 或者 Emacs。

我们首先从桌面环境开始。访问基于 GUI 的编辑器有两种方式。首先，可以从菜单系统中启动程序。这种程序一般位于“Accessories”菜单中。其次，可以使用命令行启动程序。只需打开一个终端窗口(参见第 6 章)，等待 shell 提示，然后输入程序的名称即可。在 KDE 中，基于 GUI 的文本编辑器是 **kedit**；在 Gnome 中，基于 GUI 的文本编辑器是 **gedit**。

许多 Linux 系统都拥有 KDE 或者 Gnome，但是即便您使用的不是 Linux，也可以试着运行一下 **gedit**。该程序属于 GNU 实用工具(第 2 章)，因此可以在许多不同的系统上找到这个程序。例如，从 JDS(Java Desktop System, Java 桌面系统)下的 Solaris 终端窗口中就可以运行 **gedit**。

如果没法访问基于 GUI 的编辑器，那么很有可能您的系统上会有一个基于文本的简单编辑器。大多数情况下这个编辑器是 Pico 或 Nano(这两个编辑器几乎相同，Nano 是 GNU 版本的 Pico)。为了检查系统上是否存在这些编辑器，只需查看是否可以显示它们的说明书页。如果可以显示，还可以使用说明书页自学基本使用方法：

```
man pico
man nano
```

一旦知道了如何使用一个简单的文本编辑器——无论是基于 GUI 的文本编辑器还是基于文本的 Nano/Pico 文本编辑器，就可以使用这个文本编辑器创建和编辑初始化文件和注销文件。但是，记住我前面说的话：这样的程序只适合于初学者。从长远来看，还是需要

学习 vi 或者 Emacs 文本编辑器。

14.5 登录 shell 和非登录 shell

在第 12 章中,我们讨论了交互式 shell 和非交互式 shell。当在 shell 提示处输入命令时,使用的是交互式 shell;当运行 shell 脚本时,使用的是非交互式 shell。为了理解如何使用初始化文件,我们需要将分析深入一步,因为交互式 shell 有两种不同的类型。

登录时启动的 shell 称为**登录 shell**。其他的交互式 shell 称为**非登录 shell**。这两种 shell 之间的区别非常重要,因为登录 shell 和非登录 shell 的初始化文件的处理方式大不相同。下面考虑几种常见的情形:

(1) 虚拟控制台和终端窗口

当使用桌面环境,例如 Gnome 和 KDE 时,获得 shell 提示的方式有两种:打开一个终端窗口或者切换到虚拟控制台(参见第 6 章)。当使用虚拟控制台时——假如通过按 <Ctrl-Alt-F1>组合键,要求进行登录。当进行登录时,就会启动一个登录 shell。另一方面,如果只是简单地打开一个终端窗口,则启动一个非登录 shell(因为没有进行登录)。

(2) 启动新 shell

在任何时候,都可以通过输入 shell 的名称启动一个新 shell。例如,假设您正在使用的 shell 是 Bash,而您希望试一试 Tcsh。为此,只需输入命令 **tcsh** 即可。新 shell 是一个非登录 shell(因为没有进行登录)。

(3) 使用远程主机

使用 **ssh**(Secure Shell, 安全 shell)可以连接到远程 Unix 主机。一旦 ssh 连接到远程主机,则必须进行登录。因此,这启动了一个登录 shell。

14.6 何时执行初始化文件

既然您已经理解了登录 shell 和非登录 shell 之间的区别,下面我们讨论当新 shell 启动时会发生什么事情。这里需要回答的重要问题是:执行哪些初始化文件?何时执行?

这里有两条通用规则,两条规则之间有细小的差别。这两条规则是:

- (1) 登录 shell 执行登录文件和环境文件。
- (2) 非登录 shell 只执行环境文件。

下面是具体的细节,首先从 Bourne shell 家族的成员开始。为了方便,我将使用 **\$ENV** 表示名称存储在 **\$ENV** 环境变量中的文件。例如,对于 Korn shell,可以将 **\$ENV** 的值设置为 **.kshrc**。

Bash(默认模式)

- 登录 shell: **.bash_profile**
- 非登录 shell: **.bashrc**

Bash(POSIX 模式)

- 登录 shell: **.profile**, 然后**\$ENV**
- 非登录 shell: **\$ENV**

Korn shell

- 登录 shell: **.profile**, 然后**\$ENV**
- 非登录 shell: **\$ENV**

在介绍 C-Shell 家族之前, 先补充几个注释。首先, 我们可以将所有的 Bourne shell 分成两组。3 种 shell 中有两种 shell 遵循 POSIX 规范, 使用 **.profile** 和 **\$ENV**。例外的就是默认模式中的 Bash。这是一个十分重要的预兆, 因为一旦使用 shell 一段时间后, 您将会发现 Bash 有些与众不同。

Bash 反映了在 20 世纪 90 年代中期和末期——即 Linux 和开放源代码运动的发展期间(参见第 2 章)——成熟的年青程序员的态度。从情感上讲, 开放源代码程序员有点类似于反叛者, 他们反对商业 Unix 惯例, 例如专有软件和受限的许可证协议。基于这一原因, 他们选择创建一种增强的 shell, 而不是克隆标准的 POSIX shell^{*}。这就是为什么作为一名 Bash 用户, 会遇到许多与其他 shell 表现不同的情形的原因。

例如, 只有 Bash 的登录 shell 只执行登录文件, 而不执行环境文件。对于其他所有的 shell 来说, 登录 shell 既执行登录文件, 也执行环境文件, 并按照这个顺序执行。这意味着 Bash 用户为了强制执行环境文件, 必须在登录文件中放置一条特殊的命令(稍后我将示范如何完成这一步)。

下面我们看看 C-Shell 家族的成员如何使用初始化文件:

C-Shell

- 登录 shell: **.cshrc**, 然后**.login**
- 非登录 shell: **.cshrc**

Tcsh

- 登录 shell: **.tcshrc**, 然后**.login**
- 非登录 shell: **.tcshrc**

(出于向后兼容性考虑, 如果 Tcsh 查找不到 **.tcshrc**, 它会查找 **.cshrc**。)

除了一个有趣的补充规则之外, 该模式比较直接。在 C-Shell 家族中, 登录 shell 首先执行环境文件。在 Bourne shell 家族中, 登录 shell 首先执行登录文件。为了理解为什么会出现这种情况, 我们需要讨论一下初始化文件的历史。

14.7 shell 初始化文件的历史简介

(在本节中, 我准备讨论一点 shell 的历史。有关 shell 历史的更多内容, 请参见第 11 章。)

最初的 Unix shell 由 Ken Thompson 编写, 并在 20 世纪 70 年代初期在贝尔实验室中使

^{*} 可以确定的是, Bash 程序员并不是彻底的叛逆者。他们也创建了 POSIX 模式, 以便在要求与社区标准严格一致的情形中使用。

用。这个 shell 并没有使用标准化的初始化文件。在 20 世纪 70 年代中期，贝尔实验室的程序员编写了两种新的替代 shell：Mashey shell(也称为 PMB shell)和 Bourne shell。一个名叫 Dick Haight 的程序员在 Mashey shell 中添加了对初始化文件(**.profile**)的支持。后来，Bourne shell 中也添加了这一特性。

.profile 文件只在登录时执行一次。1987 年，当 Bill Joy 在加利福尼亚大学伯克利分校开发 C-Shell 时，他通过使用两个文件替代一个文件，增强了初始化过程。第一个文件是 **.cshrc**，每次有新 shell 启动时都运行。第二个文件是 **.login**，只在登录 shell 启动时运行。因此，**.login** 在 **.cshrc** 之后执行才有意义，因为它的工作就是运行那些在登录时需要的额外命令。

1982 年，贝尔实验室的 David Korn 为 Bourne shell 开发了另一种替代产品，即 Korn shell。Korn 采纳了 Bill Joy 使用两个初始化文件的思想，即现在所谓的环境文件(**.cshrc**)和登录文件(**.login**)。因为 Korn 在贝尔实验室工作，而贝尔实验室也是 Bourne shell 的工作间，所以他为登录文件使用了名称 **.profile**。在命名环境文件时，Korn 决定让用户自己选择环境文件的名称，即将 **ENV** 变量的值设置成环境文件的名称。通过这种方式，可以根据不同目的定义不同的环境文件。

但是，一旦做出这个决定，Korn 必须确保登录文件在环境文件之前执行。否则，用户就没有办法设置 **ENV** 变量了。

这就是为什么现在 Bourne shell 家族的 shell 首先运行登录文件，而 C-Shell 家族的 shell 首先运行环境文件的原因。实际中，这通常并不是一个大问题，但是如果出现了看上去无法解决的神秘初始化问题，记住这一点可能有助于您解决问题。

14.8 初始化文件中放置什么内容

现在可以考虑前面提出的问题了：登录文件中应该放置什么内容？环境文件中应该放置什么内容？下面是答案。

登录文件有两项任务：设置环境以及初始化工作会话。因此，登录文件中应该包含下述命令：(1)创建或者修改环境变量的命令；(2)执行所有一次性操作的命令。

也就是说，登录文件就是设置 **PATH**、**PAGER** 等变量以及使用 **umask** 设置文件创建掩码(参见第 25 章)的地方。如果登录文件是远程主机的，那么还需要使用 **stty** 命令修改键映射(参见第 7 章)。最后，还可能希望在每次登录时显示一个个人消息或者其他信息。

正如本章前面讨论的，环境自动地被所有的子进程(包括新 shell)所继承。因此，环境变量(例如 **PATH**)只需要在登录文件中设置一次。在环境文件中设置环境变量并没有意义，环境文件中设置的变量在新 shell 启动时将被复位。

环境文件有一项不同的任务：设置不能在环境中保存的自定义项，特别是 shell 选项、别名和函数。因为这些设置没有存储在环境中，所以每次启动新 shell 时必须重新创建它们。

14.9 显示、创建及编辑初始化文件

您可能早已拥有一个或多个初始化文件。在共享系统上，通常由系统管理员创建这样的文件。在自己的系统上，在创建账户时可以自动地生成这些文件。如果已经拥有这样的文件，那么您可以修改这些文件，以适合自己的需要。如果还没有这样的文件，那么您可以自己创建这些文件。

初始化文件保存在 **home** 目录(自己的个人目录，参见第 23 章)中。正如前面讨论的，所有的初始化文件都是点文件，这意味着它们的名称以一个.(点号)开头。出于参考目的，图 14-1 中包含了标准初始化文件的名称。

正如前面讨论的，通过使用 **ls -a** 命令可以显示所有的点文件(没有 **-a** 选项，**ls** 命令不会显示点文件)。如果列表太长，可以将输出发送给 **less**：

```
ls -a
ls -a | less
```

一旦知道自己拥有哪些初始化文件，就可以使用 **less** 命令查看它们的内容。下面是完成该任务的命令列表：

```
less .bash_login
less .bash_logout
less .bash_profile
less .bashrc
less .cshrc
less .kshrc
less .login
less .logout
less .profile
less .tcshrc
```

为了创建或者修改一个点文件，需要使用文本编辑器。如果您已经知道 **vi** 或者 **Emacs**，那么很好，您可以使用它们中的任意一个。如果您还不会使用这两个文本编辑器，那么您可以暂时使用前面讨论的较简单的编辑器：**kedit** 或者 **gedit**。因为这两个编辑器都是基于 GUI 的编辑器，所以必须从桌面管理器中使用它们。从虚拟终端或者连接到远程主机的 CLI 中无法使用它们。

在 GUI 中，打开一个终端窗口，然后输入编辑器的名称，后面跟着希望创建或者修改的文件名称。对于 KDE，使用 **kedit**；对于 Gnome，使用 **gedit**。例如：

```
kedit .bash_profile
gedit .bash_profile
```

运行上述命令后将打开一个新窗口。如果文件早已存在，则加载这个文件，并允许修改它的内容。如果文件还不存在，则打开一个空窗口，并允许创建这个文件。

14.10 shell 脚本注释

刚才讨论的初始化文件实际上是 shell 脚本——用 shell 语言编写并且由 shell 执行的程序。稍后，我们将示范一些 shell 脚本的例子。在这之前，先大体上介绍一些与 shell 脚本和程序相关的重要内容。

请看图 14-3、14-4、14-5 和 14-6 中的初始化文件。尽管它们各不相同，但是它们之间有一个共同点：许多行以一个#(井号)字符开头。这样的行称为注释。

当执行脚本时，shell 忽略所有的注释。这就允许您在脚本中放置一些帮助记忆逻辑和理解脚本的注释。如果您以前没有编过程序，可能认为在为自己编写脚本时，注释是没有必要的。的确，当您未来阅读脚本时，能够记得各条命令中隐含的逻辑。毕竟，您就是编写脚本的人。

然而事实的真相是，尽管您的推理现在十分清晰，但是当您几天之后阅读脚本或者程序时，您很有可能记不清楚当时是怎么想的。这就是为什么所有有经验的程序员都在他们编写的东西中放置大量注释的原因。

此外，他人也有可能需要阅读您的脚本。在这种情况下，注释就更加宝贵。我的建议是在做任何事情时，记录下正在做的事情。并且，要保证您所编写的任何东西，其他人都能够理解。相信我，随后您将会发现，您永远不会为添加注释所花费的时间而感到遗憾*。

在 shell 脚本中，注释的实际定义是一个#字符，以及这一行上#字符之后的所有内容。因此，注释可以占用一整行，或者一行的一部分。考虑下述例子：

```
# Display the time, date, and current users
date; users
```

第一行是注释；第二行包含两条命令，没有注释。当 shell 执行这两行时，它将忽略注释，并且运行第二行上的命令。现在考虑下述例子：

```
date; users # Display time, date, and current users
```

在这个例子中，我们只有一行，其中包含两条命令，后面是注释。当 shell 执行这一行时，它将运行 **date** 和 **users** 命令，并且忽略这一行上的其他所有内容。因此，当编写 shell 脚本时，添加注释的方式有两种：在一行的末尾或者单独一行。

在下面几节中，我们将仔细分析几个初始化文件。在分析这几个文件的过程中，考虑一下，如果没有注释，阅读这些文件将会是多么的困难。

* 当我是位于圣地亚哥的加利福尼亚大学的一名研究生时，我是系统编程课程(一门非常技术性的编程课程)的一名助教。同时，该课程还有一名助教 Peter。

尽管 Peter 和我相处融洽，但是我们之间有一点绝对不一致。我觉得我们应该教学生在程序中大量使用注释。而 Peter 不喜欢注释，所以他就不教学生在程序中使用注释。他说注释阻挡了他阅读程序(我们需要对程序进行打分)。

在毕业之后，我去了一所医科学学校，并成为了一名专业作家。时至今日，我已经编写了 32 本书，销量总计超过 2 000 000 本，并且还翻译成多种语言。我和我漂亮、聪明的妻子生活在南加利福尼亚州，住在一个靠近海洋的房子里。我们的家庭恩爱，享受令人愉快的友谊、各种各样的兴趣以及成就。而另一方面，Peter 被证明完全失败。

我认为这一寓意是显而易见的。

14.11 Bourne shell 家族：初始化文件示例

在接下来的两节中，将讨论 4 个初始化文件的例子。您将看到的東西就是将前面 3 章中学习的知识组合在一起的结果，具体内容包括：交互式 shell、环境变量、shell 变量、shell 选项、元字符、引用、搜索路径、命令替换、历史列表、命令行编辑、别名以及注释。我的意图就是让您以这些文件为基础，根据自己的需要进行必要的修改和添加。

在本节中，我们将讨论适合于 Bourne shell 家族(Bash、Korn shell)成员的初始化文件。在下一节中，我们将讨论适合于 C-Shell 家族成员的初始化文件。无论您现在使用哪一种 shell，我希望您将两节内容都好好地看一看。在使用 Unix 和 Linux 的生涯中，您将会发现自己要使用许多种系统，阅读这两节内容有助于您熟悉这两种 shell 家族的初始化文件。

本节的目标就是仔细地分析一个登录文件示例和一个环境文件示例。登录文件位于图 14-3 中。这里要提醒您，每次登录时这个文件都会自动地执行。对于 Bash 来说，登录文件命名为**.bash_profile** 或者**.bash_login**。对于 Korn shell 或者 POSIX 模式中的 Bash 来说，登录文件命名为**.profile**。

```
# =====
# Bourne Shell family: Sample login file
# =====

# 1. Environment variables
export HISTSIZE=50
export PAGER=less
export PATH="${PATH}:/bin"
export VISUAL=vi

# 2A. Shell prompt - Bash
export PS1="(\\w) `basename ${SHELL}`[\\!]$ "

# 2B. Shell prompt - Korn Shell
export PS1="(\\$PWD) `basename ${SHELL}`[\\!]$ "

# 3. File creation mask
umask 077

# 4. Terminal settings (for remote host only)
stty erase ^H
# stty erase ^?

# 5. Display welcome message
echo "Welcome Harley."
echo "Today is `date`."
echo

# 6. System information
echo "Last three logins: "; last `logname` | head -3
echo
echo "Current users: `users`"
echo
echo "System uptime: "; uptime
echo

# 7A. Environment file - Bash
if [ -f ${HOME}/.bashrc ]
then source ${HOME}/.bashrc
fi

# 7B. Environment file - Korn Shell
export ENV=${HOME}/.kshrc

# 8. Logout file - Korn Shell
trap '. ${HOME}/.logout; exit' EXIT
```

图 14-3 Bourne shell 家族：登录文件示例

登录文件是每次登录时自动执行的文件。这个例子是一个适合于 Bash 或者 Korn shell 使用的登录文件。您可以使用这个文件作为模板，并根据自己的需要进行修改。详情请参见正文。

环境文件示例位于图 14-4 中。这个文件在每次新 shell 启动时执行。对于 Bash 来说，环境文件的名称为 **.bashrc**。对于 Korn shell 或者 POSIX 模式中的 Bash 来说，通过设置 **ENV** 变量可以将环境文件命名为任意名称。我的建议是 Korn shell 使用 **.kshrc**，Bash 使用 **.bashrc**。

```
# =====
# Bourne Shell family: Sample environment file
# =====

# 1. Shell options
set -o ignoreeof
set -o emacs
set -o noclobber

# 2. Aliases
alias a=alias
alias d=date
alias del='fc -s ls=rm'
alias h=history
alias l='ls -F'
alias la='ls -a'
alias ll='ls -l'
alias r='fc -s'

#3. Functions
# functions go here
```

图 14-4 Bourne shell 家族：环境文件示例

环境文件就是新 shell 启动时自动执行的文件。这个例子是一个适合于 Bash 和 Korn shell 使用的环境文件。您可以使用这个文件作为模板，根据自己的需要进行修改。详情请参见正文。

在 Bourne shell 家族中，登录文件在环境文件运行之前先运行，因此我们先讨论登录文件(在 C-Shell 家族中，环境文件先运行)。

现在，您应该能够理解示例文件中使用的大部分命令、变量和选项。下面我们分节进行讨论，出于参考目的，讨论过程中将给出查找更多相关信息的章号。

登录文件的第 1 节定义环境变量。将历史列表的大小设置成 50 行，默认的分页程序设置为 **less**，默认的文本编辑器是 **vi**。另外还在搜索路径的末尾添加了一个特定的目录(第 12 章：变量。第 13 章：历史列表、搜索路径。第 21 章：分页程序)。

第 2A 和 2B 节通过设置 **PS1** 环境变量定义 shell 提示。第 2A 节是针对 Bash 的，第 2B 节是针对 Korn shell 的。您或者使用第 2A 节，或者使用第 2B 节，但是不能同时使用这两节(第 12 章：变量。第 13 章：shell 提示)。

第 3 节设置文件创建掩码，控制新创建文件的默认权限(第 25 章：文件权限、**umask**)。

第 4 节只用于远程主机上的登录文件。当在自己的计算机上使用 Unix 或者 Linux 时不需要这一节。**stty** 命令为 **erase** 信号设置键映射。这里给出了两条可能的命令，您可以使用最适合于自己键盘的那一条命令(第 7 章：**erase** 信息、**stty**)。

第 5 节显示一个欢迎消息。可以将这个消息修改成自己希望的消息(第 8 章：**date**。第 12 章：**echo**、命令替换)。

第 6 节显示系统的有趣信息(第 4 章：**last**。第 8 章：**whoami**、**users**、**uptime**。第 12 章：**echo**、命令替换。第 15 章：管道线。第 16 章：**head**)。

第 7A 和 7B 节确定环境文件的运行。第 7A 节是针对 Bash 的，它查看命名为 `.bashrc` 的文件是否存在。如果存在，就告诉 shell 运行这个文件。第 7B 节是针对 Korn shell 的，它将 `ENV` 环境变量的值设置为环境文件的名称。您可以或者使用第 7A 节，或者使用第 7B 节，但是不能同时使用这两节(第 12 章：变量。第 14 章：环境文件)。

第 8 节只针对 Korn shell。默认情况下，Korn shell 不支持注销文件，注销文件是每次注销时自动执行的文件。但是，通过捕获 `EXIT` 信号可以模拟一个注销文件，`EXIT` 信号是注销时生成的信号。这里，我们指定当 `EXIT` 信号发生时，应该执行名为 `.logout` 的文件(第 7 章：捕获信号。第 14 章：注销文件)。

对于 Bourne shell 家族来说，环境文件要比登录文件简单，这是因为登录文件完成大部分的工作。环境文件只需要重新生成新 shell 启动时丢失的任何东西，包括 shell 选项、别名和函数。图 14-4 就是一个环境文件的例子。

第 1 节设置 shell 选项。`ignoreeof` 选项要求使用 `logout` 或者 `exit` 命令进行注销。通过封闭 `eof` 信号，可以防止由于不小心按了 `^D` 而注销系统(对于 Bash 来说，可以设置 `IGNOREEOF` 环境变量，而不是使用 shell 选项)。第二个 shell 选项将 Emacs 设置成命令行编辑器。严格地讲，不需要设置这个选项，因为 Emacs 就是默认的编辑器。但是，我希望明确地对它进行设置，从而可以作为一个提醒。最后，设置的是 `noclobber` 选项，从而防止重定向标准输出时不小心删除文件的内容(第 7 章：封闭 `eof` 信号。第 12 章：shell 选项、命令行编辑。第 15 章：重定向标准输出)。

第 2 节设置别名。这些别名包括 `alias` 和 `date` 的缩写、`ls` 命令的几种变体、帮助避免删错文件的 `del` 别名以及帮助使用历史列表的 `r` 和 `h` 命令(第 8 章：`date`。第 12 章：历史列表、别名。第 24 章：`ls`)。

第 3 节是为定义 shell 函数保留的。函数允许通过编程创建自己的定制命令。学习编写这样的程序已经超出了本书的范围。但是，如果您想使用函数，那么这里就是定义函数的位置。

14.12 C-Shell 家族：初始化文件示例

在本节中，将讨论 C-Shell 家族(C-Shell、Tcsh)的初始化文件。这里学习的内容与前 3 章中的内容紧密相关。我的意图就是使您可以以这些文件为基础，根据自己的需要进行必要的修改和添加。

我们从图 14-5 中的环境文件开始入手。这里要提醒您，当新 shell 启动时这个文件就会自动执行。对于 C-Shell 来说，这个文件被命名为 `.cshrc`。对于 Tcsh 来说，这个文件可以命名为 `.tcshrc` 或者 `.cshrc`。在讨论了环境文件之后，我们将讨论图 14-6 中所示的登录文件。登录文件是每次登录时执行的文件。在两种 shell 中，登录文件都被命名为 `.login`。


```
# =====
# C-Shell family: Sample environment file
# =====

# 1. Shell variables
set filec # only necessary for C-Shell
set history = 50
set ignoreeof
set noclobber
set path = (${path} ${HOME}/bin)
set savehist = 30

# 2A. Shell prompt - C-Shell
set prompt = "($PWD) `basename ${SHELL}` [\!]% "
alias cd 'cd \!* && set prompt = "($PWD) `basename ${SHELL}` [\!]\!% "'

# 2B. Shell prompt - Tcsh
set prompt = "`basename ${SHELL}` [\!]> "
set rprompt = "({~)"

# 3. Aliases
alias a alias
alias d date
alias del 'rm \!ls:*'
alias h history
alias l 'ls -F'
alias la 'ls -a'
alias ll 'ls -l'
```

图 14-5 C-Shell 家族：环境文件示例

环境文件是每次新 shell 启动时自动执行的文件。这个示例是一个适合于 C-Shell 或者 Tcsh 使用的环境文件。您可以使用这个文件作为模板，根据自己的需要进行修改。详情请参见正文。

```
# =====
# C-Shell family: sample login file
# =====

# 1. Environment variables
setenv PAGER less
setenv VISUAL vi

# 2. Command line editor - Tcsh
bindkey -e

# 3. File creation mask
umask 077

# 4. Terminal settings (for remote host only)
stty erase ^H
# stty erase ^?

# 5. Display welcome message
echo "Welcome Harley."
echo "Today is `date`."
echo

# 6. System information
echo "Last three logins: "; last `whoami` | head -3
echo
echo "Current users: `users`"
echo
echo "System uptime: "; uptime
echo
```

图 14-6 C-Shell 家族：登录文件示例

登录文件是每次登录时自动执行的文件。这个例子是一个适合于 C-Shell 或者 Tcsh 使用的登录文件。您可以使用这个文件作为模板，根据自己的需要进行修改。详情请参见正文。

我们首先讨论环境文件,这是因为在 C-Shell 家族中,它在登录文件运行之前先运行(在 Bourne shell 家族中,登录文件先运行)。

现在,您应该能够理解示例文件中使用的大部分命令、变量和选项。下面我们分节进行讨论,出于参考目的,讨论过程中将给出查找更多相关信息的章号。

环境文件的任务就是重新生成新 shell 启动时所丢失的任何东西,包括 shell 选项、shell 提示和别名。

第 1 节定义 shell 变量。我们设置了历史列表的大小,在搜索路径的末尾添加了一个特定的目录,并且打开了文件名自动补全(对于 Tcsh 来说,文件名自动补全特性默认是打开的。但是,对于 C-Shell 来说,该特性默认是不打开的,因此如果需要文件名自动补全特性,必须设置 **filec** 变量)。我们还设置了 **ignoreeof** 来封闭 **eof** 信号。这强制我们使用 **logout** 命令注销,从而防止由于不小心按下 **^D** 而注销系统。我们还设置了 **noclobber**,从而防止在重定向标准输出时不小心删除文件的内容。最后,我们设置了 **savehist**,从而在注销时可以保存历史列表(第 7 章:封闭 **eof** 信号。第 12 章:变量。第 13 章:历史列表、搜索路径、自动补全。第 15 章:重定向标准输出)。

第 2A 和 2B 节通过设置 shell 变量 **prompt** 定义 shell 提示。第 2A 节是针对 C-Shell 的;第 2B 节是针对 Tcsh 的。您可以或者使用第 2A 节,或者使用第 2B 节,但是不能同时使用这两节。注意 C-Shell 的提示中包含有工作目录的名称。因为这个值不会自动更新,所以我们定义了一个别名,当使用 **cd**(change directory, 改变目录)命令时将重置 shell 提示。对于 Tcsh 而言,这并不是必需的。对于 Tcsh,我们设置了一个帮助性的提示(**rprompt**),显示工作目录的名称(第 12 章:变量。第 13 章:shell 提示)。

第 3 节设置别名。这些别名包括 **alias** 和 **date** 的缩写、**ls** 命令的几种变体、帮助避免删错文件的 **del** 别名以及帮助使用历史列表的 **r** 和 **h** 命令(第 8 章: **date**。第 12 章:历史列表、别名。第 24 章: **ls**)。

对于 C-Shell 家族来说,登录文件要比环境文件简单,这是因为环境文件完成大多数工作(这与 Bourne shell 家族相反)。图 14-6 就是一个登录文件的例子。

登录文件的第 1 节定义环境变量。我们将默认的分页程序设置为 **less**,默认的文本编辑器设置为 **vi**(第 12 章:变量。第 21 章:分页程序)。

第 2 节将 Emacs 设置为命令行编辑器。严格地讲,不需要设置这个选项,因为 Emacs 就是默认的编辑器。但是,我希望明确地对它进行设置,从而可以作为一个提醒。因为 C-Shell 不支持命令行编辑,所以本部分只针对 Tcsh(第 12 章:命令行编辑、**bindkey**)。

第 3 节设置文件创建掩码,控制新创建文件的默认权限(第 25 章:文件权限、**umask**)。

第 4 节只用于远程主机上的登录文件。当在自己的计算机上使用 Unix 或者 Linux 时不需要这一部分。**stty** 命令为 **erase** 信号设置键映射。这里给出了两条可能的命令,您可以使用最适合于自己键盘的那一条命令(第 7 章: **erase** 信号、**stty**)。

第 5 节显示一个欢迎消息。可以将这个消息修改成自己希望的消息(第 8 章: **date**。第 12 章: **echo**、命令替换)。

第 6 节显示系统的有趣信息(第 4 章: **last**。第 8 章: **whoami**、**users**、**uptime**。第 12 章: **echo**、命令替换。第 15 章:管道线。第 16 章: **head**)。

14.13 练习

1. 复习题

1. 什么是初始化文件？列举两种类型的初始化文件。什么是注销文件？
2. 什么是点文件？什么是 **rc** 文件？
3. 什么是登录 **shell**？什么是非登录 **shell**？为什么它们之间的区别十分重要？
4. 假设您有一些自己喜欢的别名，您决定将这些定义添加到某个初始化文件中。那么应该将这些别名添加到哪个文件中呢？是登录文件还是环境文件？原因是什么？这个文件中还应该包含什么内容？

2. 应用题

1. 仔细地查看您的 **home** 目录，确定该目录下是否早已拥有一个登录文件和一个环境文件(可以使用 **ls -a** 命令列举点文件)。如果存在这些文件，那么看一看每个文件中有什么内容(使用 **less** 命令，或者在文本编辑器中打开这些文件)。

2. 使用图 14-3 或者图 14-6 中的示例文件作为模板为自己创建(或者修改)一个登录文件。如果已有一个登录文件，则在文本编辑器中打开它并进行任何修改之前，以一个不同的名称复制该登录文件。通过这种方式，当不小心犯错时，能够将登录文件恢复到最初的版本。

3. 使用图 14-4 或者图 14-5 中的示例文件作为模板为自己创建一个环境文件。如果已有一个环境文件，则根据前面练习中描述的方法对该文件做个备份。

4. 为自己创建一个注销文件。如果不确定在这个文件中放入什么内容，可以使用 **echo** 命令显示再见。这个文件的名称根据使用的 **shell** 命名(参见图 14-1)。如果使用的是 Korn **shell**，则必须捕获 **EXIT** 信号(参见本章和图 14-3 中的解释)。

3. 思考题

1. POSIX 标准要求 **shell** 支持登录文件和环境文件，但是没有要求支持注销文件。这暗含着注销文件没有其他两个文件重要。为什么是这种情况呢？

2. 在许多系统上，当创建新账户时，会自动地为新用户标识分配一个默认的登录文件，有时候还会分配一个默认的环境文件。为什么说这是一个不错的思想？您建议新用户修改这些文件还是保留这些文件不变呢？

标准 I/O：重定向和管道

从一开始，Unix 命令行就具备一些特殊的東西，使它区别于其他操作系统。这些特殊的“东西”就是所谓的 Unix 工具箱：每种 Unix 和 Linux 系统都拥有的大量程序。这些程序使用起来既简单又精巧。

在本章中，将解释 Unix 工具箱之后隐藏的设计准则。然后，我们将示范如何将基本的构建块组合成适合于自己的功能强大的工具。在第 16 章中，我们将对一些最重要的程序给予综述，并介绍日常工作可用的资源。当阅读完这两章内容之后，您就踏上了探讨最有趣和最享受的计算机技能之路。

15.1 Unix 设计准则

20 世纪 60 年代，成为 Unix 第一批开发人员的贝尔实验室研究人员在一种称为 Multics 的操作系统上工作(参见第 1 章)。Multics 操作系统存在的问题之一就是太过笨重。Multics 设计团队试图使他们的系统完成太多的事情，从而满足大多数人们。当 1969 年刚开始设计 Unix 时，只有两名开发人员，他们强烈地感觉到最重要的是要避免 Multics 操作系统和其他类似操作系统的复杂性。

因此，他们采用了斯巴达人的思想，认为表述的经济性是最重要的，要放在首位。他们的观点是：每个程序都应该是一个单独的工具，或许还有几个基本的选项。一个程序应该只做一件事，但必须出色地完成这件事情。如果需要执行一项复杂的任务，应该通过对现有工具的组合来完成(当存在可能性时)，而不是再去编写一个新程序。

例如，几乎所有的 Unix 程序都生成某些类型的输出。当程序显示大量的输出时，数据很快输出以至于大部分输出来不及阅读就已滚动出屏幕。一种解决方法就是要求每个程序在需要时都能每次一屏地显示输出。这种解决方法正是最初 Unix 开发人员希望避免的。为什么所有的程序都要集成同一个功能呢？难道就没有一种比较简单的方法来确保程序的输出以适合用户阅读的方式呈现给用户吗？

就此而言，为什么每个程序都应该知道其输出去往何处呢？有时候您可能希望将输出显示在屏幕上；有时候您可能希望将输出保存在一个文件中；甚至有时候还希望将输出发送到另一个程序中，做进一步处理。

基于这些原因, Unix 设计人员创建了一个单独的工具, 其任务就是每次一屏地显示数据。这个工具就是 **more**, 因为在显示了一屏幕数据之后, 程序会显示一个提示--More--告诉用户还有更多的数据。

这个工具的使用非常简单。用户阅读完一屏幕数据之后, 按<Space>键就会显示下一屏幕数据。当用户希望结束时, 键入 **q** 就可以退出。

在 **more** 程序设计出来之后, 程序员们就不用担心他们程序的输出该如何显示了。如果您是一名程序员, 那么您知道无论何时, 当用户在运行程序的过程中发现输出内容太多时, 他会将程序的输出发送给 **more**(本章后面将介绍它的使用方法)。如果您是一名用户, 那么您知道, 不管使用多少个程序, 您只需学习使用一种输出显示工具。

即便是在今天, 这种方法还有 3 方面重要的优点。首先, 当设计 Unix 工具时可以保持工具的简单性。例如, 无需再赋予新程序每次一屏的显示功能, 因为已经有一种工具可以完成这一功能。同理, 还有排序输出、移除特定列、删除没有包含特定模式的行等工具(参见第 16 章)。因为这些工具已对所有的 Unix 用户可用, 所以不需要在自己的程序中包含这样的功能。

这就引出了第二个优点。因为每个工具只需要完成一件事情, 所以程序员就可以集中自己的精力完成这一件事情。如果您在设计一个在数据文件中搜索特定模式的程序, 那么您可以尽最大的可能使这个程序成为最好的模式搜索程序; 如果您在设计一个排序程序, 那么您可以尽最大的可能使这个程序成为最好的排序程序; 等等。

第三个优点就是易于使用。作为一名用户, 一旦学会了控制标准屏幕显示工具的命令, 就可以知道如何控制任意程序的输出。

因此, Unix 的设计准则可以用两句话概括:

- 每个程序或者命令应该是一个工具, 它只完成一件事情, 但一定要完成好这件事情。
 - 当需要新工具时, 最好对现有的工具进行组合, 而不是编写一个新工具。
- 有时候将这一设计准则描述为:
- “Small is beautiful(小的就是完美的)”或者“Less is more(少的就更好)”。

15.2 Unix 新设计准则

因为 Unix 已经风风雨雨经历了 40 多年, 所以有必要问一下, 从长远来看, Unix 的设计准则是否是成功的? 答案是: 也成功也不成功。

在很大程度上, Unix 设计准则依然如初。正如第 16 章所示, Unix 中有大量作用单一的工具, 当有需求时可以方便地将它们组合在一起。

此外, 因为最初的 Unix 开发人员设计得如此之好, 所以 30 多年以前的程序现在可以与全新设计的程序无缝地工作。与 Windows 或者 Macintosh 世界比较一下就可以知道做到这一点多么不易。

但是, 最初的设计准则在 3 个重要方面被证实是不合适的。首先, 许多人禁不住创建新版本的基本工具。这意味着有时候为了完成同一个工作, 必须学习使用多个工具。

例如,多年以来,有3个屏幕显示程序经常使用:**more**、**pg**和**less**。现在,大多数人使用**less**,它是3个程序中功能最强大(并且最复杂)的程序。但是,**more**程序的使用比较简单,而且所有系统上都有这个程序,所以您必须学习使用这个程序。否则,有一天,您登录到一台使用**more**显示输出的系统,如果您只知道**less**,就会遇到麻烦。另一方面,只理解**more**也还不够,因为在许多系统上,**less**是默认的程序(而且**less**程序更加出色)。所以底线是:您需要学习至少两种不同的屏幕显示程序。

Unix 设计准则中不适合的第二个方面就是无法完全胜任对用户日益增长的需求的处理。“小的就是完美的”这一思想有很大的吸引力,但随着用户越来越成熟,他们的需求也越来越高,很明显,简单的工具经常不够用。

例如,最初的 Unix 文本编辑器是 **ed**(该名称代表“editor(编辑器)”,发音是两个单独的字母“ee-dee”)。**ed**在将输出打印到纸张上的终端中使用。**ed**程序只有少数几条命令,它的使用非常简单,可以快速地学会。如果您在早期使用过 Unix,那么您会发现 **ed** 程序是一个朴实的、不矫揉造作的工具:它只做一件事(文本编辑),而且它做得非常出色*。

随着编辑器的发展,**ed**也在尽量地完善。但是,没过几年时间,终端得到了极大的改进,Unix 用户的需求也越来越高。为了响应这些需求,程序员们开始开发新的编辑器。实际上,几年下来,程序员们开发的编辑器几乎有数十种。

对于主流用户来说,**ed**被一个称为 **ex** 的程序取代(**ex**的名称代表“extended editor(扩展编辑器)”,发音是两个单独的字母“ee-ex”)。然后,**ex**本身被扩展,成为了 **vi**(“visual editor(可视编辑器)”,发音为“vee-eye”)。作为 **ed/ex/vi** 家族的备选编辑器,人们又开发了一个称为 Emacs 的完全不同的编辑系统。

现在,**vi**和 Emacs 是最流行的 Unix 文本编辑器,但是没有一个人曾经说过它们简单和朴实。实际上,**vi**(参见第22章)和 Emacs 都相当复杂。

原有的 Unix 设计准则中不适合的第三个方面是对处理 CLI(命令行界面)的基本限制。众所周知,CLI 是基于文本的。这意味着它无法处理图形和图像,或者没有包含纯文本的文件(例如电子表格或者字处理程序文档)。

大多数命令行程序读取和写入文本,这就是为什么这样的程序可以集成在一起工作的原因:它们都使用相同类型的数据。但是,这意味着当希望使用其他类型的数据——非文本数据时,必须使用其他类型的程序。这就是我在第5章和第6章中指出,必须同时学习使用 CLI 和 GUI 环境的原因。

基于这些原因,必须认真掌握学习 Unix 的方法。在1979年,当 Unix 面世只有10年时,Unix 的原始设计准则仍然如初,而且可以学习几乎所有常见命令。现在,Unix 中有许多东西要学,您不可能知道所有东西,即便是大部分也不可能。这意味着必须有选择地学习希望学习的程序和工具。此外,在自学使用工具的过程中,还必须有选择地学习希望掌握的选项和命令。

在阅读接下来的两章内容时,有一些重要的事需要记住。不管怎么样,在阅读过程中应该坐在计算机的前面,遇到新命令时就输入新命令。这就是学习使用 Unix 的方式。但是,

* **ed** 编辑器在所有的 Unix 和 Linux 系统上仍然可用。您有时间时可以试一下,但首先要阅读一下 **ed** 的说明书页(**man ed**)。

希望您不仅仅能够记住细节。在阅读和体验过程中，希望您有自己的观点，时不时地回头来看看并问自己：“我现在正在学习的工具适合于什么应用呢？”

我的目标就是您能及时理解所谓的 Unix 新设计准则：

- “除非程序无法更小，否则小的就是完美的。”

提示

无论何时，当学习如何使用新程序时，不要试图记忆每个细节。正确的做法是回答下面的 3 个问题：

- (1) 这个程序可以做什么？
- (2) 这个程序有什么基本细节？也就是说，哪些细节在大多数时间适合大多数人？
- (3) 需要时可以在什么地方查找到更多的帮助？

15.3 标准输入、标准输出和标准错误

如果说要有效地使用 Unix 就要掌握一个最基本的概念，那么这个概念就是标准输入和标准输出。理解了这个概念，在成为 Unix 高手的路途上就前进了巨大的一步。

这里的基本思想很简单：每个基于文本的程序都应该能够从任何源接受输入，并向任何目标写入输出。

例如，假设您拥有一个对文本行进行排序的程序。您可以选择从键盘键入文本、从现有文件中读取文本，甚至使用另一个程序的输出。同样，`sort` 程序应该能够在屏幕上显示输出、将输出写入文件或者将输出发送给另一个程序做进一步处理。

使用这样的程序有两个突出的优点。首先，用户会具有极大的灵活性。当运行程序时，可以根据自己的需要定义输入和输出(I/O)，这意味着每项任务只需学习一个程序。例如，排序少量数据并将数据在屏幕上显示的程序，也可以排序大量的数据并将数据保存到文件中。

第二个优点就是以这种方式处理 I/O 使新工具的创建简单多了。当编写程序时，可以依靠 Unix 处理输入和输出，这意味着不必再担心输入和输出的各种情况。因此，可以集中精力设计和编写工具。

这里的关键就是输入的源和输出的目标不由程序员指定。程序员以一种通用的方式编写程序的读取和写入。以后，当程序真正开始运行时，`shell` 就会将程序连接到用户希望使用的输入和输出上*。

为了实现这一思想，Unix 的开发人员设计了一种读取数据的通用方法(称之为**标准输入**)和两种写出数据的通用方法(称之为**标准输出**和**标准错误**)。有两种不同输出目标的原因在于标准输出用于正常输出，而标准错误用于错误消息。综合起来，我们称这些功能为**标准 I/O**(standard I/O，发音为“standard eye-oh”)。

* 从历史观点上说，使用抽象 I/O 设备的思想是为了允许程序员编写的程序与具体的硬件无关。您能看出在哲学上，该思想与第 5 章讨论的抽象层次以及第 7 章讨论的终端描述数据库(Termcap 和 Terminfo)如何相关吗？

实际中，我们通常非正式地称这3个术语，就好像它们是实际对象一样。因此，我们可以说：“To save the output of a program, write standard output to a file(为了保存程序的输出，需要将标准输出写到文件中)。”其实我们的真正意思是：“为了保存程序的输出，需要告诉 shell 将输出目标设置成文件。”

理解标准输入、标准输出和标准错误的概念对熟练地使用 Unix 非常关键。此外，其他编程语言(例如 C 和 C++)中也使用相同的概念控制 I/O。

提示

标准输入、标准输出和标准错误通常缩写为 `stdin`、`stdout` 和 `stderr`。当在对话中使用这些缩写时，它们的发音分别为“standard in”、“standard out”和“standard error”。

例如，如果您正在创建文档资料，您可能会写：“The `sort` program reads from `stdin`, and writes to `stdout` and `stderr`(`sort` 程序从 `stdin` 读取输入，向 `stdout` 和 `stderr` 写入输出)。”如果您对听众朗读这段话，那么您应该如下发音：“The `sort` program reads from standard in, and writes to standard out and standard error.”

15.4 重定向标准输出

在登录时，shell 会自动地将标准输入设置为键盘，将标准输出和标准错误设置为屏幕。这意味着，默认情况下，大多数程序从键盘读取输入，并将输出写入到屏幕。

但是，每次输入命令时，都可以告诉 shell 在此命令执行期间重置标准输入、标准输出或标准错误，这就是真正体现 Unix 强大功能的地方。

实际上，可以告诉 shell：“我希望运行 `sort` 命令，并将输出保存在一个叫 `names` 的文件中。在执行该命令时，我希望将标准输出写入到这个文件中。当这条命令结束后，我希望将标准输出重新设置回屏幕。”

下面解释其工作方式：如果希望命令的输出写到屏幕上，则无需做任何事情。这是自动的。

如果希望将命令的输出写入到文件中，在命令的后面键入 `>`(大于号)字符，后面跟着文件的名称即可。例如：

```
sort > names
```

该命令将把它的输出写入到一个叫 `names` 的文件中。`>`字符的使用比较适当，因为它看上去就像一个箭头，指示了输出的路径。

当以这种方式将命令的输出写入到文件中时，这个文件可以存在，也可以不存在。如果这个文件不存在，那么 shell 将自动地创建这个文件。在我们的例子中，shell 将创建一个叫 `names` 的文件。

如果这个文件已经存在，那么它的内容将被替换，所以必须小心。例如，如果文件 `names` 已经存在，那么 `names` 文件中的原始内容将永久丢失。

在一些情况中，这种方式不错，因为您确实想替换这个文件的内容(用较新的信息)。

但在另一些情况中，可能不希望丢失文件中原有的内容。更确切地说，就是希望将新数据添加到原有的数据后面。实现这一目的需要使用>>字符，即连续两个大于号字符。这告诉 shell 将新数据追加在已有文件的尾部。因此，考虑下述命令：

```
sort >> names
```

如果文件 **names** 不存在，shell 将创建这个文件。如果这个文件已经存在，新数据将追加到这个文件的尾部。这个文件不会丢失任何内容。

当将标准输出发送给文件时，我们称之为**重定向**标准输出。因此，在前面的两个例子中，我们将标准输出重定向到 **names** 文件。

现在您能够明白为什么有两种类型的输出了：标准输出和标准错误。如果将标准输出重定向到一个文件，您不会错过任何错误消息，因为它们仍然显示在显示器上。

当重定向输出时，一定要小心，不要丢失了重要的数据。有两种方法可以做到这一点。第一种方法是每次将输出重定向到文件时，仔细地考虑：希望替换这个文件的当前内容吗？如果希望替换，则可以使用>。或者，希望将新数据追加到这个文件的末尾吗？如果是这种情况，则使用>>。

第二种方法采用了一种防护措施，即告诉 shell 永远不替换现有文件的内容。通过设置 **noclobber** 选项(Bash、Korn shell)或者 shell 变量 **noclobber**(C-Shell、Tcsh)可以完成这一点。我们将在下一节中讨论。

15.5 防止文件被重定向替换或创建

在上一节中，我们指出当使用>将标准输出重定向到文件时，文件中已有的任何数据都将丢失。我们还指出当使用>>向文件中追加输出时，如果这个文件不存在，则会创建这个文件。

有时候，可能不希望 shell 代表自己做这样的假设。例如，假设您有一个 **names** 文件，这个文件中包含有 5000 行数据。您希望将 **sort** 命令的输出追加到这个文件的末尾。换句话说，就是希望输入下述命令：

```
sort >> names
```

但是，您犯了一个错误，一不小心输入了：

```
sort > names
```

这会发生什么情况呢？所有的原始数据都被清除。而且，数据还被快速地清除。即便在按下<Return>键时注意到了错误，即便立即按下^C 键来终止程序(通过发送 **intr** 信号，参见第 7 章)，也都太晚了。文件中的数据永远不见了。

下面解释原因。只要按下<Return>键，shell 就会通过删除目标文件的内容为 **sort** 程序做好准备。因为 shell 比人快多了，所以当意识到要终止程序时，目标文件早已清空了。

为了避免这样的灾难发生，可以告诉 shell 当使用>重定向输入时，不要替换已存在的

文件。另外, 当使用 C-Shell 家族的 shell 时, 也可以告诉 shell 在使用>>追加数据时不要创建新文件。这样就确保了文件不会被意外地替换或创建。

为了让 shell 替我们做这些预防措施, 需要设置所谓的 **noclobber** 功能。对于 Bourne shell 家族(Bash、Korn shell)来说, 需要设置 shell 选项 **noclobber**:

```
set -o noclobber
```

复位这个选项, 可以使用:

```
set +o noclobber
```

对于 C-Shell 家族(C-Shell、Tcsh)来说, 需要设置 shell 变量 **noclobber**:

```
set noclobber
```

复位这个变量, 可以使用:

```
unset noclobber
```

(有关选项和变量的讨论请参见第 12 章, 摘要信息请参见附录 G。)

一旦设置了 **noclobber**, 就会拥有内置的防护。例如, 假设您已经拥有一个 **names** 文件, 并且输入了:

```
sort > names
```

您将看到一个错误消息, 告诉您文件 **names** 已经存在。下面是 Bash 给出的一条这样的消息:

```
bash: names: cannot overwrite existing file
```

下面是 Tcsh 中一条等价的消息:

```
names: File exists.
```

在这两个例子中, shell 都拒绝执行命令, 所以文件将是安全的。

如果您确实希望替换这个文件呢? 这种情况下, 可以临时忽略 **noclobber**。对于 Bourne shell 来说, 这时需要使用>|来取代>:

```
sort >| names
```

对于 C-Shell 来说, 需要使用>!来取代>:

```
sort >! names
```

通过使用>|或者>!取代>, 就可以告诉 shell 即使文件存在, 也要重定向标准输出。

正如前面讨论的, 通过使用>>取代>重定向标准输出可以将数据追加到文件中。不管是哪一种情况, 如果输出文件不存在, shell 将创建这个文件。但是, 如果要追加数据的话, 那么看上去您希望这个文件早已存在。因此, 如果使用>>而文件不存在的话, 则有可能犯错。这里设置 **noclobber** 有帮助吗?

Bourne shell 家族不可以。如果在设置了 **noclobber** 选项的情况下,在 Bash 或 Korn shell 中追加数据,处理方式依旧。C-Shell 和 Tcsh 就比较好,它们会告诉您文件不存在,并且拒绝执行命令。

例如,假设您是一名 C-Shell 或者 Tcsh 用户,shell 变量 **noclobber** 已经设置,并且有一个命名为 **addresses** 的文件,您希望在这个文件中追加数据。您输入了命令:

```
sort >> address
```

您将看到一个错误消息:

```
address: No such file or directory
```

此时您可能会说:“噢,我应该键入 **addresses**,而不是 **address**。谢谢您,C-Shell 先生。”

当然,有时候您可能希望向文件中追加数据,而且希望忽略 **noclobber**。例如,您是一名 C-Shell 用户,而且出于安全考虑,已经设置了 **noclobber**。您希望对 **input** 文件进行排序,并将数据追加到文件 **output** 中。

如果 **output** 文件不存在,那么您希望创建这个文件。这里重要的是,如果 **output** 文件已经存在,那么您不希望丢失这个文件中已有的任何内容,所以要使用追加(>>),而不是替换(>)。如果还没有设置 **noclobber**,则可以使用:

```
sort >> output
```

因为已经设置了 **noclobber**,所以必须忽略它,即使用>>!取代>>:

```
sort >>! output
```

这样将仅忽略这条命令的自动检查。

15.6 重定向标准输入

默认情况下,标准输入被设置成键盘。这意味着,当运行需要读取数据的程序时,程序期望用户通过键盘输入数据,每次一行。当结束输入数据时,可以按[^]D(<Ctrl-D>组合键)发送 **eof** 信号(参见第 7 章)。按下[^]D 表示已经没有数据了。

下面举一个例子,大家可以试一试。输入:

```
sort
```

sort 程序现在等待从标准输入(键盘)输入数据。键入一些数据行,例如,输入:

```
Harley
Casey
Weedly
Linda
Melissa
```


当在最后一行上按下<Return>键之后，按[^]D 发送 eof 信号。sort 程序将根据字母顺序对数据进行排序，并将结果写入到标准输出中。默认情况下输出是屏幕，所以您将看到：

```
Casey
Harley
Linda
Melissa
Weedly
```

但是，有许多时候，可能希望重定向标准输入，使其从文件中读取数据，而不是从键盘读取。为此，只需在命令的最后键入<(小于号)，后面跟着文件的名称即可。

例如，为了排序 **names** 文件中所包含的数据，可以使用命令：

```
sort < names
```

可以看出，<字符是一个很好的选择，因为它看上去就像一个示范输入路径的箭头。

下面举一个例子，大家可以试一试。正如第 11 章中所述，每个用户标识的基本信息包含在文件/etc/passwd 中。输入下述命令可以对这个文件进行排序：

```
sort < /etc/passwd
```

正如您想象的，标准输入和标准输出重定向可以同时指定，而且这样做还十分频繁。考虑下述例子：

```
sort < rawdata > report
```

这个命令从 **rawdata** 文件中读取数据，排序之后，将输出写入到 **report** 文件中。

15.7 文件描述符、Bourne shell 家族重定向标准错误

尽管下面的讨论面向 Bourne shell 家族，但是我们也讨论与 Unix I/O 相关的重要思想。基于这一原因，不管您现在使用哪一种 shell，我都希望您仔细阅读这一整节的内容。

正如前面所述，shell 提供两种不同的输出目标：标准输出和标准错误。标准输出用于正常输出，标准错误用于错误消息。默认情况下，两种类型的输出都显示在屏幕上。但是，当需要时，可以将这两种输出流分开。

如果选择分离这两种输出流，则有多种方法可供选择。例如，可以将标准输出重定向到文件，并保存到文件中。同时，保持标准错误不变，这样就不会错过任何错误消息(将显示在屏幕上)。另外，还可以将标准输出重定向到一个文件，而将标准错误重定向到另一个文件。或者，将标准输出和标准错误都重定向到一个文件。

另外，还可以将标准输出或者标准错误(或者两者同时)发送给另一个程序做进一步处理。当在本章后面讨论管道线时，将示范这种方法。

在两种不同的 shell 家族中，重定向标准错误的语法并不相同。我们首先讨论 Bourne shell 家族，然后再讨论 C-Shell 家族。在开始之前，首先对 Unix 处理 I/O 的过程进行解释。

在 Unix 进程中，每个输入源和每个输出目标都由一个唯一的数字标识，这个数字称为文件描述符(file descriptor)。例如，一个进程可能从文件#8 中读取数据，并将数据写入到文件#6 中。在编写程序时，使用文件描述符控制 I/O，每个文件使用一个文件描述符。

在 Bourne shell 家族中，重定向输入或输出的正式语法是在文件描述符数字之后使用 <(小于号)或>(大于号)。例如，假设一个程序 **calculate** 被设计为将输出写入到文件描述符为 8 的文件中。使用下述命令运行该程序，并将输出重定向到 **results** 文件中：

```
calculate 8> results
```

默认情况下，Unix 为每个进程提供 3 个预定义的文件描述符，而且大多数时候这已经够用。默认的文件描述符是 0 代表标准输入，1 代表标准输出，2 代表标准错误。

因此，在 Bourne shell 家族中，重定向标准输入的语法是使用 0<，后面跟着输入文件的名称。例如：

```
command 0< inputfile
```

其中 *command* 是命令，*inputfile* 是文件的名称。

重定向标准输出和标准错误的语法与此相似。对于标准输出，语法是：

```
command 1> outputfile
```

对于标准错误，语法是：

```
command 2> errorfile
```

其中 *command* 是命令，*outputfile* 和 *errorfile* 是文件的名称。

为了方便起见，如果在重定向输入时省略了 0，那么 shell 假定指的就是标准输入。因此，下述两条命令是等价的：

```
sort 0< rawdata
sort < rawdata
```

同理，如果在重定向输出时省略 1，那么 shell 假定指的就是标准输出。因此，下述两条命令也是等价的：

```
sort 1> results
sort > results
```

当然，在同一条命令中可以使用不止一个重定向。在下面的例子中，**sort** 命令从 **rawdata** 文件中读取输入，将输出写入到 **results** 文件中，并且将错误消息写入到 **errors** 文件中：

```
sort 0< rawdata 1> results 2> errors
sort < rawdata > results 2> errors
```

注意只能省略标准输入和标准输出的文件描述符。对于标准错误来说，必须包含 2。这可以通过下述简单的例子示范，在这个例子中，标准错误被重定向到 **errors** 文件：

```
sort 2> errors
```

当重定向标准错误时, 它不影响标准输入和标准输出。在这个例子中, 标准输入仍然来自键盘, 而标准输出仍然显示在显示器上。

和所有的重定向一样, 当将标准错误写入到已经存在的文件中时, 新数据将替换文件中已有的数据。在最后一个例子中, 文件 **errors** 中的内容将丢失。

如果希望将新输出追加到文件的末尾, 可以使用 **2>>** 取代 **2>**。例如:

```
sort 2>> errors
```

在 C-Shell 家族中重定向标准错误比较复杂。在讨论之前, 我们需要先讨论一个称为子 shell(subshell)的重要功能。即便您现在不使用 C-Shell 或者 Tcsh, 我也希望您阅读下一节内容, 因为子 shell 对每个人都十分重要。

15.8 子 shell

为了解子 shell 的概念, 需要首先了解一些 Unix 进程的概念。在第 26 章中, 我们将详细地讨论 Unix 进程。现在, 我们只给一个简单的概要。

进程就是加载到内存中并且准备运行的程序, 以及程序的数据和跟踪程序所需的信息。当进程需要启动另一个进程时, 这个进程就创建一个副本进程。原始的进程称为**父进程**, 副本进程称为**子进程**。

子进程开始运行后, 父进程等待子进程死亡(也就是结束)。一旦子进程死亡, 父进程就会被唤醒, 重新获得控制权并再次开始运行, 此时子进程消失。

为了将这一思想与日常工作相关联, 可以考虑输入命令时发生的事情。shell 解析命令, 判断这条命令是内部命令(shell 中内置的命令)还是外部命令(单独的程序)。当输入的是内部命令时, shell 就会在自己的进程中直接解释命令。这种情况下不需要创建新的进程。

当输入的是外部命令时, shell 查找合适的程序, 然后以一个新进程运行这个程序。当该程序终止时, shell 重新获得控制权, 并等待输入另一条命令。在这个例子中, shell 就是父进程, shell 运行的程序就是子进程。

下面考虑当启动一个全新 shell 时发生的事情。例如, 如果使用的是 Bash, 并输入 **bash** 命令(或者使用的是 C-Shell, 并输入 **cs** 命令, 等等)。

原始的 shell(父进程)启动一个新 shell(子进程)。当一个 shell 启动另一个 shell 时, 我们称第二个 shell 为**子 shell**。因此, 我们可以说, 无论何时, 当启动一个新 shell(通过输入 **bash**、**ksh**、**cs** 或者 **tcsh**)时, 都会创建一个子 shell。现在输入的所有命令都由子 shell 解释。结束子 shell 时, 可以按 **^D** 发送 **eof** 信号(参见第 7 章)。此时, 父 shell 重新获得控制权。接下来, 无论输入什么命令都由原始的 shell 解释。

当创建子 shell 时, 它继承父 shell 的环境(参见第 12 章)。但是, 子 shell 对环境的任何改变都不会传递回父 shell。因此, 如果子 shell 修改或者创建了环境变量, 改变并不会影响原始的 shell。

这意味着，在子 shell 中，可以按照自己的意愿做任何事情，而不影响父 shell。这种能力非常便利，所以 Unix 提供了两种使用子 shell 的方法。

第一种，正如前面所述，可以通过输入命令明确地启动一个全新的 shell。例如，如果使用的是 Bash，则可以输入 **bash**。然后就可以按照自己的意愿做任何事情，而不影响原始的 shell。例如，如果修改了一个环境变量或者一个 shell 变量，那么在输入 **^D** 之后，该修改将立即消失；也就是说，在新 shell 死亡的时刻，原始 shell 重新获得控制权。

有时候，可能希望在子 shell 中运行一小组命令，或者就一条命令，但又不希望启动一个全新的 shell。Unix 为这种情况提供了一个特殊的工具：将命令括在圆括号中。这样就可以告诉 shell 在子 shell 中运行命令。

例如，为了在子 shell 中运行 **date** 命令，可以使用：

```
(date)
```

当然，在子 shell 中运行 **date** 没有任何理由。但是，下面是一个非常现实的例子，该例子使用目录进行说明。

在第 24 章中，我们将讨论目录，它是用来包含文件的。您可以创建任意数量的目录，并且在工作时，从一个目录切换到另一个目录中。在任何时间，当前工作所处的目录称为工作目录。

假设您有两个目录 **documents** 和 **spreadsheets**，而且当前位于 **documents** 目录中。您希望改变到 **spreadsheets** 目录，运行 **calculate** 程序。在运行这个程序之前，需要将环境变量 **DATA** 设置为包含特定原始数据的文件的名称。在这个例子中，这个文件就是 **statistics**。一旦程序运行完毕，就需要将 **DATA** 恢复成原来的值，并改变回 **documents** 目录(换句话说，需要将环境重置回原来的状态)。

一种方法就是启动一个新 shell，然后改变工作目录，修改 **DATA** 的值并运行 **calculate** 程序。一旦做完这些，就可以通过按 **^D** 退出新 shell。当新 shell 结束时，旧 shell 重新获得控制权，工作目录和变量 **DATA** 将恢复为原始状态。

假定您使用的 shell 是 Bash，下面是具体的操作(**cd** 命令将在第 24 章中讨论，它用来改变工作目录。现在还不用关心它的语法)。

```
bash
cd ../spreadsheets
export DATA=statistics
calculate
^D
```

下面是一种更简单的方法，该方法使用圆括号：

```
(cd ../spreadsheets; export DATA=statistics; calculate)
```

当通过这种方式使用子 shell 时，就不用关心新 shell 的启动和停止了。这些会自动完成。此外，在子 shell 中，可以按照自己的意愿对环境进行任何改变，而且没有永久的影响。例如，可以改变工作目录、创建或者修改环境变量、创建或者修改 shell 变量、改变 shell 选项等。

有时候会发现位于圆括号中的这些命令被称为一个**编组(grouping)**，特别是阅读 C-Shell

家族的文档资料时。在我们的例子中，使用了一个含有3条命令的编组。使用编组和子 shell 的最常见的原因是防止 **cd**(change directory, 改变目录)命令影响当前的 shell。其通用格式为：

```
(cd directory; command)
```

15.9 在 C-Shell 家族中重定向标准错误

对于 Bourne shell 家族来说，标准错误的重定向相当简单。只需在文件的名称之前使用 **2>**即可。对于 C-Shell 家族(C-Shell、Tcsh)来说，标准错误的重定向就没有那么简单，这是因为有一个有趣的限制，稍后我将详细介绍。

对于 C-Shell 家族来说，重定向标准错误的基本语法为：

```
command >& outputfile
```

其中 *command* 是命令，*outputfile* 是文件的名称。

例如，如果您使用的是 C-Shell 或者 Tcsh，那么下面的命令将把标准错误重定向到 **output** 文件：

```
sort >& output
```

如果希望在已有文件的末尾追加输入，可以使用 **>>&**取代 **>&**。在下面的例子中，输出就追加到文件 **output** 的末尾：

```
sort >>& output
```

如果已经设置了 shell 变量 **noclobber**(本章前面解释过)，而且希望临时重写这个变量，则可以使用 **>&!**替代 **>&**。例如：

```
sort >&! output
```

在这个例子中，文件的内容将被替换，即便是设置了 **noclobber**。

那么前面提及的限制是什么呢？当使用 **>&**或者 **>\$!**时，shell 将同时重定向标准输出和标准错误。实际上，对于 C-Shell 家族来说，没有简单的方法可以单独重定向标准错误。因此，在上一个例子中，标准输出和标准错误都被重定向到 **output** 文件。

碰巧的是有一种方法可以使标准错误和标准输出的重定向相互分离。但是，这样做需要先了解如何使用子 shell(本章前面解释过)。语法为：

```
(command > outputfile) >& errorfile
```

其中 *command* 是命令，*outputfile* 和 *errorfile* 是文件的名称。

例如，假设您使用的是 **sort** 程序，希望将标准输出重定向到 **output** 文件，将标准错误重定向到 **errors** 文件。可以使用：

```
(sort > output) >& errors
```

在这个例子中，**sort** 在子 shell 中运行，并且在这个子 shell 中，标准输出被重定向了。在这个子 shell 之外，剩下的一个输出——标准错误——被重定向到一个不同的文件。所获

得的结果就是将每种类型的输出重定向到自己的文件。

当然，如果希望，还可以使用>>和>>&将输出追加到文件的末尾。例如，为了将标准输出追加到 **output** 文件的末尾，并且将标准错误追加到 **errors** 文件的末尾，可以使用类似于下面的命令：

```
(sort >> output) >>& errors
```

15.10 组合标准输出和标准错误

所有的 shell 都支持重定向标准输出和标准错误。但是，如果希望将标准输出和标准错误重定向到同一个位置，该怎么办呢？

对于 C-Shell 家族来说，这比较容易，因为只要使用>&(替代)或者>>&(追加)，shell 就会自动地组合两种输出流。例如，在下面的 C-Shell 命令中，标准输出和标准错误都被重定向到 **output** 文件中：

```
sort >& output
sort >>& output
```

对于 Bourne shell 家族来说，情形就比较复杂。我们将讨论具体细节，然后再示范一个在 Bash 中使用的捷径。

基本思想就是将一种类型的输出重定向到一个文件，然后再将另一种类型的输出重定向到同一个位置。相关语法为：

```
command x> outputfile y>&x
```

其中 *command* 是命令，*x* 和 *y* 是文件描述符，*outputfile* 是文件的名称。

例如，在下面的 **sort** 命令中，标准输出(文件描述符 1)被重定向到 **output** 文件。然后标准错误(文件描述符 2)被重定向至和文件描述符 1 相同的位置。整体的结果是将正常输出和错误消息都发送给同一个文件：

```
sort 1> output 2>&1
```

既然文件描述符 1 是默认的重定向输出，所以可以省略数字 1 的第一个实例：

```
sort > output 2>&1
```

在继续之前，先讨论一个很容易犯的有趣错误。如果将重定向的顺序反过来会发生什么情况呢？

```
sort 2>&1 > output
```

尽管看上去和上面的例子几乎相同，但是这条命令不能正常工作。下面解释具体原因：

指令 **2>&1** 告诉 shell 将文件描述符 2(标准错误)的输出发送给和文件描述符 1(标准输出)相同的位置。但是，在这个例子中，该指令在标准输出被重定向之前发送给了 shell。因此，当 shell 处理 **2>&1** 时，标准输出仍然发送给显示器(默认情况)。这意味着标准错误最终被重定向到显示器，这也是它默认发送的位置。

最终获得的结果是标准错误发送到显示器，而标准输出发送给文件(仔细地考虑一下，确保理解这一点)。

如果希望同时重定向标准输出和标准错误，但是希望将输出追加到文件的末尾，该怎么办呢？只需使用>>取代>：

```
sort >> output 2>&1
```

在这个例子中，使用>>将致使标准输出和标准错误都追加到文件 **output** 的末尾。

您可能会问，有没有可能在组合两种类型的输出时以标准错误为先呢？也就是说，是不是可以将标准错误重定向到一个文件，然后再将标准输出发送到相同位置呢？答案是肯定的：

```
sort 2> output 1>&2
sort 2>> output 1>&2
```

该命令与前面的例子有所不同，但是它们拥有相同的结果。

可以看出，Bourne shell 家族在组合两种类型的输出流时比较复杂。能不能使它简单一些呢？难道不能直接将标准输出和标准错误发送给同一个文件吗？例如：

```
sort > output 2> output
```

尽管这看上去似乎可以正常工作，但是实际上它不能，因为，如果在一条命令中重定向向同一个文件两次的话，一个重定向就会覆盖另一个重定向。

下面介绍一条捷径。Bourne shell 家族的所有成员都可以使用上述技巧，特别是 Bash 和 Korn shell。但是，对于 Bash，可以使用&>或者>&(选择自己喜欢的一种)同时重定向标准输出和标准错误：

```
sort &> output
sort >& output
```

这样就避免了记忆较复杂的模式。但是，如果希望既重定向标准输出又重定向标准错误，并且还追加输出，则需要使用上面讨论的模式：

```
sort >> output 2>&1
```

现在，如果您只是一个普通人，那么您可能会感觉迷惑。不要担心。在前面几节中讨论的所有内容都在图 15-1 和图 15-2(参见本章后面)中进行了归纳。我的经验就是要多加练习，然后就会发现重定向规则非常容易记忆。

15.11 抛弃输出

为什么希望将输出抛弃呢？

有时候，您需要运行一个程序，因为它执行特定的动作，但是您并不关心这个程序的输出。有时候，可能希望查看程序的正常输出，但是不关心程序的错误消息。在第一种情况中，可以抛弃标准输出；在第二种情况中，可以抛弃标准错误。

这样做时，只需重定向标准输出，将它发送给一个特殊的文件/dev/null(该名称的发音是“slash-dev-slash-null”，尽管有时候可能听到别人说“dev-null”)。在阅读了第 23 章中有关 Unix 文件系统的讨论之后，您就会理解名称/dev/null 的含义。关于/dev/null 文件，一件重要的事情就是发送给它的任何东西都会永远消失。当 Unix 人士聚在一起时，有时候您也可能听到/dev/null 被奇怪地称为位桶(bit bucket)。

例如，假设您有一个程序 **update**，该程序读取并修改大量的数据文件。在其工作时，**update** 显示发生事件的统计信息。如果不希望看到这个统计信息，则可以将标准输出重定向到/dev/null:

```
update > /dev/null
```

同理，如果希望查看正常输出，但是不希望查看错误消息，可以重定向标准错误。对于 Bourne shell 家族(Bash、Korn shell)，可以使用：

```
update 2> /dev/null
```

对于 C-Shell 家族(C-Shell、Tcsh)，可以使用：

```
update >& /dev/null
```

正如前面所述，上述 C-Shell 命令重定向了标准输出和标准错误，从而抛弃了所有的输出。在 Bourne shell 家族的 shell 中，可以使用下述命令抛弃所有输出：

```
update > /dev/null 2>&1
```

那么如果您使用的是 C-Shell，您希望抛弃标准错误，但是又不希望抛弃标准输出，该怎么办呢？可以使用我们在讨论将标准错误和标准输出重定向到不同文件时所说的技术。在那个例子中，我们在子 shell 中运行下述命令：

```
(update > output) >& errors
```

这样做就可以将两个输出流分开。使用相同的方法，我们可以将标准错误重定向到/dev/null，从而抛弃标准错误。同时，将标准输出重定向到/dev/tty，以保存标准输出：

```
(update > /dev/tty) >& /dev/null
```

特殊文件/dev/tty 表示终端，我们将在第 23 章中详细讨论。现在，您只需知道当把输出发送给/dev/tty 时，输出就显示在显示器上。通过这种方式，我们就可以使 C-Shell 和 Tcsh 将标准输出发送到显示器，而将标准错误抛弃。^{*}

15.12 重定向：小结和体验

重定向标准输入、标准输出和标准错误非常简单。但是，各种变体容易引起混淆。不过，我的目标是您应该理解两种 shell 家族的所有变体，这需要大量的实践。为了使这一过程更简单，我将从两个方面提供帮助。

^{*} 如果您在想：“为什么要这么麻烦地做这样简单的事情？”那么您是正确的。这无疑是 C-Shell 家族的失败。然而，这种环境下我们还能实现它，是不是更酷？

首先，出于参考目的，图 15-1 和图 15-2 提供了所有重定向元字符的摘要信息。图 15-1 针对的是 Bourne shell 家族，图 15-2 针对的是 C-Shell 家族。在这些摘要中，您将发现我们已经讨论过的所有特性。另外还有管道的引用。也就是说使用一个程序的输出作为另一个程序的输入，下一节将讨论管道。

元字符	动作
<	重定向标准输入(同 0<)
>	重定向标准输出(同 1>)
>	重定向标准输出；强制重写
>>	追加标准输出(同 1>>)
2>	重定向标准错误
2>>	追加标准错误
2&>1	将标准错误重定向到标准输出
>&或者&>	重定向标准输出+标准错误(只适用于 Bash)
	将标准输出通过管道传送给另一条命令
2>&1	将标准输出+标准错误通过管道传送给另一条命令

图 15-1 Bourne shell 家族：标准 I/O 的重定向

大多数命令行程序使用标准 I/O 进行输入和输出。输入来源于标准输入(stdin)，正常输出发送到标准输出(stdout)，错误消息则发送给标准错误(stderr)。

对于 Bourne shell 家族，可以使用文件描述符(stdin=0、stdout=1、stderr=2)及各种元字符控制标准 I/O。当没有歧义性时，可以省略文件描述符。为了防止不小心重写了已有的文件，可以设置 shell 选项 **noclobber**。如果设置了 **noclobber** 选项，则可以使用>|强制进行重写。详情请参见正文。

元字符	动作
<	重定向标准输入
>	重定向标准输出
>!	重定向标准输出；强制重写
>&	重定向标准输出+标准错误
>&!	重定向标准输出+标准错误；强制重写
>>	追加标准输出
>>!	追加标准输出；强制文件创建
>>&	追加标准输出+标准错误
>>&!	追加标准输出+标准错误；强制文件创建
	将标准输出通过管道传送给另一条命令
&	将标准输出+标准错误通过管道传送给另一条命令

图 15-2 C-Shell 家族：标准 I/O 的重定向

对于 C-Shell 家族来说，可以使用各种不同的元字符控制标准 I/O。为了防止不小心重写已有的文件或者创建一个新文件，可以设置 shell 变量 **noclobber**。如果设置了 **noclobber** 变量，则可以使用一个!字符强制进行重写或者文件创建。注意，与 Bourne shell 家族(图 15-1)不同，在不重定向标准输出的情况下没有简单的方法单独重定向标准错误。详情请参见正文。

第二种帮助就是提供许多例子，以便于进行体验。为了体验标准输出和标准错误，需要一个既能生成正常输出又能提供错误消息的简单命令。我发现的最好的这类命令是 **ls** 的一个变体。

ls(list, 列举)命令显示文件的信息，我们将在第 24 章中正式讨论该命令。使用 **-l(long, 长)**选项，**ls** 可以显示文件的信息。

这里的思想就是使用 **ls -l** 显示两个文件 **a** 和 **b** 的信息。文件 **a** 已存在，但是文件 **b** 不存在。因此，我们将看到两种类型的输出：标准输出将显示文件 **a** 的信息，标准错误将显示一个说明文件 **b** 不存在的错误消息。然后就可以使用这个样例命令练习标准输出和标准错误的重定向。

在开始之前，必须创建文件 **a**。可以使用 **touch** 命令来创建文件。我们将在第 25 章中讨论 **touch** 命令。现在，您只需知道如果对一个不存在的文件使用 **touch** 命令，那么 **touch** 命令将用这个文件名创建一个空文件。因此，如果文件 **a** 不存在，则可以使用下述命令创建它：

```
touch a
```

现在可以使用 **ls** 命令显示文件 **a**(存在)和文件 **b**(不存在)的信息：

```
ls -l a b
```

下面是一个典型输出：

```
b: No such file or directory
-rw----- 1 harley staff 0 Jun 17 13:42 a
```

第一行是标准错误。它包含一个错误消息，告诉我们文件 **b** 不存在。第二行是标准输出。它包含有文件 **a** 的信息(注意文件名在这一行的末尾)。现在不必关心细节问题，我们将在第 24 章中详细讨论。

现在就可以体验示例命令了。请看图 15-1 和图 15-2，选择一些元字符进行练习。作为示例，我们将标准输出重定向到文件 **output**：

```
ls -l a b > output
```

当运行这条命令时，不会看到标准输出，因为输出已经发送到文件 **output** 中。但是，还会看到标准错误：

```
b: No such file or directory
```

为了查看文件 **output** 的内容，可以使用 **cat** 命令(我们将在第 16 章中讨论 **cat** 命令)。

```
cat output
```

在这个例子中，**cat** 将显示 **output** 文件的内容，即前面命令的标准输出：

```
-rw----- 1 harley staff 0 Jun 17 13:42 a
```

下面再举一个例子。假设您使用的是 **Bash**，且希望将标准输出和标准错误重定向到两

个不同的文件:

```
ls -l a b > output 2> errors
```

因为所有的输出都被重定向,所以在屏幕上看不到任何东西。为了查看标准输出,可以使用:

```
cat output
```

为了查看标准错误,可以使用:

```
cat errors
```

在体验过程中,可以使用 **rm**(remove, 移除)命令删除文件。例如,为了删除文件 **output** 和 **errors**, 可以使用:

```
rm output errors
```

当结束体验时,可以使用下述命令删除文件 **a**:

```
rm a
```

现在我们已经有一个好的样本命令(**ls -l a b**),并且知道如何显示一个短文件的内容(**cat filename**),剩下的就是练习了。

我的建议是为图 15-1 和图 15-2 中每种类型的输出重定向创建至少一个例子^{*}。尽管这样需要花费一定的时间,但是一旦完成这些体验,您将比世界上 99.44% 的 Unix 用户知道更多的重定向知识。

提示

为了体验重定向,我们使用了 **ls** 命令的一种变体:

```
ls -l a b > output
ls -l a b > output 2> errors
```

为了使体验更加简单,可以用一个简单的名称为该命令创建一个别名(参见第 13 章)。对于 Bourne shell 家族(Bash、Korn shell),可以使用:

```
alias x='ls -l a b'
```

对于 C-Shell 家族(C-Shell、Tcsh),可以使用:

```
alias x 'ls -l a b'
```

^{*} 是的,对于两个 shell 家族,我希望您至少练习每类家族中的一种 shell。如果不知道使用哪一种 shell,可以使用 Bash 和 Tcsh。

如果您平时使用 Bash,则可以试一试这些例子,然后输入 **tcsh** 命令启动一个 Tcsh shell,并再次试一试这些例子。如果平时使用 Tcsh,则可以首先使用这个 shell,然后再输入 **bash** 命令启动一个 Bash shell。

不管现在您使用哪一种 shell,您永远也不知道未来会使用什么 shell。所以不管使用哪一种 shell,我都希望您理解基本的 shell 概念:环境变量、shell 变量、选项和重定向。

一旦拥有了这样的别名，测试命令将变得更加简单：

```
x > output
x > output 2> errors
x >& output
```

这是一个值得记住的技术。

15.13 管道线

在本章前面讨论 Unix 设计准则时，我们解释到初期 Unix 开发人员的目标就是构建小的工具，每个工具只做一件事情，但是一定要出色地完成。他们的意图就是当用户面对靠一个工具无法解决的问题时，能够使用一组工具来完成这个任务。

例如，假设您为政府部门工作，您有 3 个大文件，这 3 个文件中包含有全国所有聪明人的信息。在每个文件中，每个人对应有一行信息，包括这个人的姓名。您的问题就是查找有多少人叫 Harley。

如果把这个问题交给一名有经验的 Unix 人士，那么他知道如何去做。首先，他使用 **cat**(catenate, 连接)命令将这几个文件组合在一起。然后，使用 **grep** 命令抽取所有包含单词 Harley 的行。最后，使用 **wc**(word count, 单词统计)命令和 **-l**(line count, 行统计)选项统计行的数量。

下面我们讨论如何基于前面所学的知识组合这样一个解决方法。我们使用重定向在临时文件中存储中间结果，当工作完成时再将临时文件删除。下面是完成该任务的命令，现在先不管这些命令运转的细节问题(我们将在本书后面讨论它们)。为了帮助理解发生的事情，这里添加了一些注释：

```
cat file1 file2 file3 > tempfile1      # combine files
grep Harley < tempfile1 > tempfile2    # extract lines
wc -l < tempfile2                       # count lines
rm tempfile1 tempfile2                 # delete temp files
```

仔细地看看这些命令。在继续讨论之前，确保理解如何依靠重定向标准输出和标准输入，将数据保存在临时文件中，从而将数据从一个程序传递给另一个程序。

上面使用的命令序列可以完成任务。但是，其中还有一个缺陷：将所有事情连接在一起的粘合剂——使用临时文件进行重定向——使该解决方案难以理解。此外，太大的复杂性也使人容易犯错误。

为了使这样的解决方法简单些，shell 允许创建一序列命令，在这一序列命令中，一个程序的标准输出可以自动地发送给下一个程序的标准输入。当这样做时，两个程序之间的连接就是管道(pipe)，而命令序列本身称为管道线(pipeline)。

在创建管道线时，只需将希望键入的命令用竖线(|)字符(管道符号)分隔开即可。例如，前面一组 4 条命令可以用一个管道线替换：


```
cat file1 file2 file3 | grep Harley | wc -l
```

为了解管道线，需要从左向右阅读命令行。每次看到管道符号时，可以认为一个程序的标准输出成为下一个程序的标准输入。

管道线如此简单的原因在于 shell 负责所有的细节，从而不用使用临时文件。在我们的例子中，shell 自动地将 **cat** 命令的标准输出与 **grep** 命令的标准输入，以及 **grep** 命令的标准输出和 **wc** 命令的标准输入连接起来。

对于 Bourne shell 家族来说，可以将标准输出和标准错误组合在一起，然后一起发送给另一个程序。语法为：

```
command1 2>&1 | command2
```

其中 *command1* 和 *command2* 都是命令。

在下面的例子中，**ls** 命令的标准输出和标准错误都发送给 **sort** 命令：

```
ls -l file1 file2 2>&1 | sort
```

对于 C-Shell 家族来说，语法为：

```
command1 |& command2
```

例如：

```
ls -l file1 file2 |& sort
```

当谈论管道线时，我们通常将单词 **pipe** 作为动词使用，指从一个程序向另一个程序发送数据。例如，在第一个例子中，我们将 **cat** 的输出管道传送给 **grep**，并且将 **grep** 的输出管道传送给 **wc**。在第二个例子中，我们将 **ls** 的标准输出和标准错误都管道传送给 **sort**。

当考虑类似于上面的例子时，可以很容易想象出一幅管道线的画面：数据从一端进入，从另一端出来。但是，更好的比喻就是将其看作组装线。原始的数据从一端进入，然后一个接一个的程序对数据进行处理，直至在另一端，数据以结果形式出现。

当创建管道线时，必须使用能够从标准输入读取文本，并向标准输出写入文本的程序。我们称这样的程序为“过滤器”，许多程序都是过滤器。我们将在第 16~19 章中讨论最重要的过滤器。如果您是一名程序员，那么您可以通过编写自己的过滤器创建自己的工具。

实际中，您将会发现大多数管道线只连续使用 2 条或者 3 条命令。到目前为止，管道线最常见的应用就是将一些命令的输出传递给 **less**（参见第 21 章），从而可以每次一屏地显示命令的输出。例如，为了显示 2008 年的日历，可以使用：

```
cal 2008 | less
```

(**cal** 程序在第 8 章中解释过。)

掌握 Unix 艺术的基本技能之一就是学习何时以及如何将程序组合成管道线以解决问题。当创建管道线时，可以使用任意多的过滤器，有时候会看到管道线包含 5 个、6 个或者更多个程序，创造性地组合在一起。实际上，当构建管道线时，您只受自己的智力以及

有关过滤器的知识的限制*。

提示

当使用那些使用了管道或者重定向了标准 I/O 的命令时，<、>或|两边没必要加空格。但是，使用这样的空格绝对是个好主意。例如，最好不要这样输入：

```
ls -l a b >output 2>errors
cat f1 f2 f3|grep Harley|wc -l
```

而应该这样输入：

```
ls -l a b > output 2> errors
cat f1 f2 f3 | grep Harley | wc -l
```

以这种方式使用空格可以使键入错误的发生机率最小，并且使命令容易阅读。当编写 shell 脚本时，这尤为重要。

15.14 管道线分流：tee

有时候，可能希望将程序的输出同时发送到两个地方。例如，希望将一个输出保存在文件中，同时还发送到另一个程序。为了示范这个过程，请考虑下面的例子：

```
cat names1 names2 names3 | grep Harley
```

这个管道线的作用就是显示文件 **names1**、**names2** 和 **names3** 中包含单词“Harley”的所有行(说明：**cat** 命令将 3 个文件组合在一起，**grep** 命令抽取包含字符串“Harley”的所有行。这两条命令分别在第 16 章和第 19 章中讨论)。

假设您希望保存一份组合后的文件。换句话说，就是希望将 **cat** 的输出发送给一个文件，并且同时发送给 **grep**。

使用 **tee** 命令可以实现这一目的。**tee** 命令的作用就是从标准输入读取数据，并向标准输出和一个文件各发送一份数据。**tee** 命令的语法为：

```
tee [-a] file...
```

其中 *file* 是希望将数据发送到的文件的名称。

通常，只对 **tee** 使用一个文件名称，例如：

```
cat names1 names2 names3 | tee masterlist | grep Harley
```

在这个例子中，**cat** 的输出保存在 **masterlist** 文件中。同时，**cat** 的输出还管道传送给 **grep**。

当使用 **tee** 时，通过指定多个文件名可以为输出保存不止一份副本。例如，在下面的管道线中，**tee** 将 **cat** 的输出复制到两个文件 **d1** 和 **d2** 中：

```
cat names1 names2 names3 | tee d1 d2 | grep Harley
```

* 这应该没什么问题。在阅读完第 16~19 章之后，您就会理解如何使用最重要的过滤器。此外，作为我的读者，很明显您的智力要高于平均水平。

如果在 **tee** 命令中指定的文件不存在，那么 **tee** 命令会创建这个文件。但是，使用时必须小心，因为如果文件已经存在，那么 **tee** 命令将重写这个文件，原始的内容就会丢失。

如果希望 **tee** 命令在文件的末尾追加数据，而不是替换文件，则可以使用 **-a**(append, 追加)选项。例如：

```
cat names1 names2 names3 | tee -a backup | grep Harley
```

该命令将 **cat** 的输出保存在 **backup** 文件中。如果 **backup** 已经存在，则不会丢失任何内容，因为 **cat** 的输出将追加到文件的末尾。

当希望查看命令的输出并且同时把输出保存到文件中时，在管道线末尾使用 **tee** 命令非常便利。例如，假设您希望使用 **who** 命令(参见第8章)显示当前登录系统的用户标识。但是，您不仅希望显示信息，而且还希望将信息保存在 **status** 文件中。实现这一任务的一种方法就是使用两条单独的命令：

```
who
who > status
```

但是，通过使用 **tee**，一条命令就够了：

```
who | tee status
```

要特别注意这种模式，我希望您记住：

```
command | tee file
```

注意不必在 **tee** 之后使用另一个程序。这是因为 **tee** 将它的输出发送给标准输出，默认情况下就是屏幕。

在我们的例子中，**tee** 从标准输入读取 **who** 的输出，并将输出写入到文件 **status** 和屏幕上。如果发现输出太长，则可以将输出管道传送给 **less**，从而每次显示一屏输出：

```
who | tee status | less
```

名称含义

tee

在管道工程行业中，“tee”型连接器将两个管道以直线连接，同时还提供一个额外的出口，将水流直角转向。例如，可以使用“tee”型连接器使水由左向右流，同时还向下流。实际的连接器看上去就像一个大写字母“T”。

当使用 Unix 的命令 **tee** 时，可以把数据从一个程序移动到另一个程序想象成数据由左向右流。同时，数据的一份副本还通过“tee”型连接器的分支发送到一个文件中。

15.15 管道线的重要性

1964年10月11日，贝尔实验室的研究员 Doug McIlroy 撰写了一篇10页的内部备忘录。在这篇备忘录中他提出了许多建议和思想。备忘录的最后一页中包含了他所提思想的精华。这段话是这样的：

“To put my strongest concerns into a nutshell:

We should have some ways of connecting programs like [a] garden hose — screw in another segment when it becomes necessary to massage data in another way...”

(简要地说：我们应该有一些连接程序的方法，就像花园的水管一样，当需要以另一种方式处理数据时，强制数据进入另一段中……)

回过头来一看，我们可以发现 McIlroy 说将程序组合起来解决身边的问题应该比较容易。尽管这一思想非常重要，但是直到 5 年之后，这一思想也没有实现。

到 20 世纪 70 年代初，原始的 Unix 项目在贝尔实验室中顺利进行(参见第 2 章)。这时，McIlroy 是诞生 Unix 的研究部门的经理。他对许多研究领域都做出了重要的贡献，包括 Unix 的一些方面。例如，正是 McIlroy 要求 Unix 的说明书页应该简短、准确。

有一段时间，McIlroy 一直在促进有关输入和输出流思想的实现。但是，直至 1972 年，Ken Thompson(参见第 2 章)才在 Unix 中添加了管道线。为了添加管道功能，Thompson 被迫修改了大量已有的程序，将程序的输入源由文件修改成标准输入。

管道功能实现并设计出合适的表示法之后，管道线就成为 Unix 的集成部分，用户的创造性也越来越强。根据 McIlroy 所言，在此之后“……我们可以无节制地拥有管道线。每个人都有一个管道线。看看这个，看看那个……”

实际上，管道线的实现是 Unix 设计准则提升的催化剂。正如 McIlroy 所回忆的：“……每个人都开始提出 Unix 设计准则。编写完成一件事情且完成得很好的程序。编写一起工作的程序。编写处理文本流的程序，因为它是一个统一界面……”

现在，大约 30 年以后，Unix 的管道功能与 1972 的原始功能基本相同：这是多么了不起的成就！实际上，在很大程度上，正是管道线和标准 I/O 使 Unix 的命令行界面功能如此强大。基于这一原因，我鼓励大家学习如何熟练地使用管道线，并练习将它们应用到自己的日常工作去。

为了帮助大家踏上 Unix 的成功之路，第 16~19 章专门讨论 Unix 的过滤器，其中的原始资料可以帮助形成解决实际问题的创造性方法。

但是，在讨论过滤器之前，我们还要讨论一个主题：条件执行。

15.16 条件执行

有时候，希望在前一条命令成功执行的条件下执行另一条命令。实现这种目的的命令语法为：

```
command1 && command2
```

有时候，可能希望在前一条命令没有成功执行的条件下执行另一条命令。这种情况的命令语法为：

```
command1 || command2
```

这里的思想就是当前一条命令成功执行或者失败时才执行另一条命令，这就是所谓的

条件执行。

条件执行主要在 shell 脚本中使用。但是，有时候，在输入命令时使用条件执行也比较方便。下面举一些例子。

假设您有一个文件 **people**，这个文件中包含各种人士的信息。您希望对 **people** 的内容进行排序，并将输出保存在文件 **contacts** 中。但是，您只希望在 **people** 文件中包含“Harley”时才这样做。

首先，我们如何判断文件中是否包含名字“Harley”呢？我们使用 **grep** 命令(参见第19章)显示文件中所有包含“Harley”的行。该命令为：

```
grep Harley people
```

如果 **grep** 命令成功，它将在标准输出中显示包含“Harley”的各行。如果 **grep** 命令失败，那么它不显示任何内容。在我们的例子中，如果 **grep** 命令成功，那么我们希望运行命令：

```
sort people > contacts
```

如果 **grep** 命令不成功，则不做任何事。

下面就是使用条件执行完成这一任务的命令：

```
grep Harley people && sort people > contacts
```

尽管这条命令可以完成任务，但是它有一个小问题。如果 **grep** 命令在文件中发现任何满足标准的行，那么它将在屏幕上显示这些行。大多数时候，这可能有用，但是在本例中，我们不希望看到任何输出。我们需要的只是运行 **grep** 命令，测试它是否成功。

解决方法就是将 **grep** 的输出重定向到 **/dev/null**，从而抛弃 **grep** 的输出：

```
grep Harley people > /dev/null && sort people > contacts
```

偶尔，我们可能希望在前一条命令失败时才执行另一条命令。例如，假设您希望运行 **update** 程序，该程序将运行几分钟，处理一些事情。如果 **update** 程序成功结束，那么一切正常，没什么事可做。如果 **update** 程序没有成功结束，则您希望知道它没有成功。下述命令将在 **update** 程序失败的情况下显示一个警告消息：

```
update || echo "The update program failed."
```

提示

如果需要终止一个正在运行的管道线，则只需按下 **^C** 发送 **intr** 信号(参见第7章)。当管道线中的一个程序停止时(因为它在等待输入)，这是一种重新获得控制权的好方法。

15.17 练习

1. 复习题

1. 总结 Unix 的设计准则。
2. 在第10章中，我曾经给出在学习新程序的语法时要问的3个问题：这条命令做什

么用？如何使用选项？如何使用参数？同理，当开始学习一个新程序时应该问(以及回答)的 3 个问题是什么呢？

3. 综合来讲，术语“标准 I/O”指的是标准输入、标准输出和标准错误。分别定义这 3 个术语。它们的缩写分别是什么？重定向标准 I/O 意味着什么？示范如何重定向这 3 种类型的标准 I/O。

4. 什么是管道线？使用什么元字符分隔管道线的各个部分？在管道线的末尾使用什么程序可以每次一屏地显示输出？

5. 使用什么程序为管道线中的数据保存一份副本？

2. 应用题

1. 示范如何将 **date** 命令的标准输出重定向到 **currentdate** 文件。

2. 下述管道线统计当前登录到系统的用户标识的数量(**wc -w** 命令用来统计单词数量，参见第 18 章)。

```
users | wc -w
```

不改变管道线的输出，修改该命令，将 **users** 的输出在 **userlist** 文件中保存一份副本。

3. 口令文件(/etc/passwd)为系统中注册的每个用户标识都包含有一行。创建一个管道线，对口令文件中的各行进行排序，并将结果保存在 **userid** 文件中，然后显示系统中用户标识的数量。

4. 在下述管道线中，**find** 命令(第 25 章解释)搜索/etc 下的所有目录，以查找用户标识 **root** 拥有的文件。然后将这些文件的名称写到标准输出，每行一个名称。**find** 的输出被传送给 **wc -l** 命令以统计行的数量：

```
find /etc -type f -user root -print | wc -l
```

在 **find** 命令运行时，它将生成各种您不想看到的错误消息。您的目标就是重写管道线，在不影响其他输出的情况下抛弃错误消息。示范如何为 Bourne shell 家族实现这种方法。附带思考一下，看看能不能为 C-Shell 家族设计一种完成该任务的方法(提示：在子 shell 中使用子 shell)。

3. 思考题

1. Unix 设计准则的一个重要方面就是当需要新工具时，最好组合现有的工具，而不是编写新的工具。当试图在基于 GUI 的工具中应用这一原则时会发生什么情况？这是好还是坏呢？

2. 对于 Bourne shell 家族来说，分别重定向标准输出和标准错误非常简单。这就使有选择地保存或者抛弃错误消息非常简单。对于 C-Shell 家族来说，分别重定向这两种类型的输出比较复杂。这会有多重要？C-Shell 由 Bill Joy 设计，那个时代，他是一名出色的程序员。您认为他为什么要创建这样一个复杂的系统？

3. 通常，计算机世界的改变非常快。您认为为什么如此众多的 Unix 基本设计准则还如此适用，即便它们是在 30 年前创建的？

过滤器：简介和基本操作

在第 15 章中，我们讨论了 Unix 的设计准则如何引导众多程序的开发，每个程序都被设计成一个能够出色地完成一件事情的工具。我们还讨论了如何重定向输入和输出，以及如何创建管道线，在管道线中数据由一个程序传递给下一个程序。

在接下来的 4 章(第 16、17、18 和 19 章)中，将通过介绍 Unix 中大量的“过滤器”程序继续进行我们的讨论(稍后将示范过滤器的准确定义)。通过使用这些程序，配合第 15 章中讨论的技术，我们将构建灵活的、定制的解决方法，从而能够解决各种各样的问题。

我们首先讨论一些一般性的话题，从而帮助理解过滤器的重要性以及使用它们的方式。然后我们再讨论最重要的 Unix 过滤器，尽管一些过滤器是相关的，但是它们是独立的工具，所以不必按特定的顺序学习它们。如果希望学习一个具体的过滤器，可以直接跳到那一节中学习。但是，如果您有时间，我还是希望您按顺序从头到尾阅读所有这 4 章的内容，因为我会按这个顺序讲述不同的重要思想。如果在开始之前，您希望查看所有的过滤器列表，则可以查看图 16-1，该图位于本章后面。

在第 20 章中，将讨论一个非常重要的功能，即正则表达式，它一般用来指定模式。正则表达式可以极大地增强过滤器的功能，因此可以认为接下来的 4 章与第 20 章的内容是互补的。

16.1 命令和选项变体

第 16、17、18 和 19 章的目的是讨论基本的 Unix 过滤器。所有这些程序在大多数版本的 Unix 和 Linux 系统中都可用。如果某个程序在您的系统上没法使用，则可能是因为您的系统中默认没有安装该程序，您需要安装一个特定的程序包。例如，在一些 Linux 的发布版本中，除非安装了 **binutils**(二进制文件实用工具)程序包，否则可能无法使用 **strings** 程序(参见第 19 章)。

众所周知，特定程序的细节在各个系统之间可能有所不同。在本章中，描述的都是 GNU 版本的命令。因为 Linux 和 FreeBSD 都使用 GNU 实用工具(参见第 2 章)，所以如果您是 Linux 或者 FreeBSD 的用户，那么在这 4 章中所阅读的内容不加改变就可以应用到您的系统上。如果您其他类型的 Unix 用户，那么在这 4 章所学的内容与您的系统之间可能存在

差别，但是差别不大。

例如，在本章后面，我们将讨论 **cat** 命令可以使用的 3 个选项。如果您使用的是 Linux 或者 FreeBSD，那么这些选项和我示范的一样，它们拥有相同的含义。但是，如果您使用的是 Solaris，那么将有一个选项(**-s**)拥有不同的含义。

在讨论每个过滤器时，将介绍该程序最重要的选项。您应该明白大多数程序都还有一些我们没有讨论的选项。实际上，所有的 GNU 实用工具都有大量的选项，其中有许多选项平时可能并不需要。

在阅读本章时，请记住，无论何时，当希望学习一个程序时，都要使用 **man** 命令访问联机手册，阅读系统提供的正式文档资料，如果使用的是 GNU 实用工具，还可以使用 **info** 命令访问 Info 系统(这些都在第 9 章中解释过)。特别是，可以使用 **man** 命令显示特定命令所有可用的选项。例如，为了学习 **cat** 命令(本章讨论)，可以使用：

```
man cat
info cat
```

在开始之前，先介绍两个适用于 GNU 实用工具(Linux 和 FreeBSD 使用的实用工具)的特点。首先，正如第 10 章中讨论的，大多数 GNU 实用工具都拥有两种类型的选项。其中有短选项，包含一个-(连字符)，后面跟一个字符；还有长选项，包含两个-，后面跟一个单词。在第 10 章中，我们称之为“-”选项和“=”选项，因为大多数人也是这样谈论它们的。

通常，大多数重要的选项都是短选项。在大多数情况中，长选项或者是短选项的同义词，或者是平常不需要的深奥选项。基于这一原因，在本书中，通常只讨论短选项。

但是，有一个长选项大家应该记住。对于 GNU 实用工具来说，大多数命令都识别 **-help** 选项。使用这个选项可以显示几乎所有命令的语法，包括命令选项的摘要信息。例如，为了显示 **cat** 命令的语法和选项，可以使用：

```
cat --help
```

16.2 过滤器

在第 15 章中，我们示范了如何将几个程序按顺序组合起来，形成一个管道线，管道线几乎完全类似于组装线。例如，考虑下面的命令，在这个命令中，数据按顺序通过了 4 个程序：**cat**、**grep**、**sort** 和 **less**。

```
cat new old extra | grep Harley | sort | less
```

在这个管道线中，我们将 3 个文件 **new**、**old** 和 **extra** 组合在一起(通过使用 **cat**)，提取包含 **Harley** 的所有行(通过使用 **grep**)，然后对这些结果进行排序(通过使用 **sort**)。最后，使用 **less** 每次一屏地显示最终输出。

现在不用关心 **cat**、**grep** 和 **sort** 的使用细节。我们将在后面讨论它们。现在，我只希望您能够理解，如果一个程序设计为能够在管道线中使用是多么的有用。

我们称这样的程序为过滤器。例如，**cat**、**grep** 和 **sort** 都是过滤器。这样的程序读取数据、对数据执行一些操作，然后写入结果。更精确地讲，过滤器就是任何能够从标准输入读取文本数据并向标准输出写入文本数据(每次一行)的程序。通常，大多数过滤器都被设计成工具，出色地完成一件事情。

更有趣的是，管道线中的第一个和最后一个程序不必是过滤器。例如，在上面的例子中，我们使用 **less** 命令显示 **sort** 的输出。我们将在第 21 章中详细讨论 **less** 命令。但是现在，我可以告诉您，当 **less** 显示输出时，它允许每次一屏地查看所有的数据、向后滚动和向前滚动、搜索具体模式等。很明显，**less** 并不是每次一行地将数据写到标准输出，所以这意味着它不是过滤器。

同理，该管道线中的第一条命令 **cat**(用来组合文件)没有从标准输入读取数据。尽管 **cat** 可以用作过滤器，但是在这个例子中，它从文件中读取输入，而不是从标准输入中读取输入。因此，在这个例子中，**cat** 也不是过滤器。

提示

当经常需要使用一个具体的管道线时，可以通过在环境文件中创建一个别名，永久地定义这个管道线(参见第 14 章)。这样就可以随时使用这个管道线，不必在每次使用时键入它。

16.3 是否应该创建自己的过滤器

如果您是一名程序员，那么您就不难创建自己的过滤器，您所需要做的就是编写一个使用标准 I/O，每次一行地读取和写入文本数据的程序或者 **shell** 脚本。任何这样的程序都是过滤器，因此可以在管道线中使用。

但是，在设计自己的程序之前，我先提醒一下，每个 **Unix** 和 **Linux** 系统都提供有数百个程序，其中许多程序都是过滤器。实际上，在过去 35 年间，历史上一些最聪明的程序员已经创建并完善了许多过滤器。

这意味着，如果您想创建一个过滤器，那么其他人可能在很久以前就有这种想法并付诸实施了。实际上，我们在本章中将要讨论的大多数工具都已经有 30 多年的历史了！因此，当遇到问题时，在编写自己的程序之前，最好是先查找是否有现成的工具可用。这就是为什么我要花费大量的时间讲授如何使用联机手册和 **Info** 系统(参见第 9 章)，以及 **man**、**whatis**、**apropos** 和 **info**(同样位于第 9 章中)。

尽管能够编写程序相当便利，但是熟练地使用 **Unix** 并不需要能够编写创建新工具的程序。对于大多数人来说，熟练地使用 **Unix** 意味着能够将已有的工具组合在一起解决问题。

16.4 问题解决过程

如果您观察一名有经验的 **Unix** 人士使用过滤器创建管道线，那么您会发现这种技术非常神秘。不知道怎么回事，看上去他准确地知道使用哪个过滤器，知道如何将这些过滤

器组合在一起。最终，您也可以这样做。您需要的就是知识和练习。但是，最好首先将这一过程分解成一系列步骤，因此下面我们开始讨论如何做到这一点。

您的目标就是理解如何通过将许多过滤器组合成一个管道线来解决问题。如果需要，还可以使用不止一行命令行，甚至是包含一串命令的 shell 脚本。但是，最聪明的 Unix 人士大都使用一行命令解决他们的问题，因此这就是您的目标。

现在，除非您实际上学习了如何使用一些 Unix 过滤器，否则您能做的事情并不多。因此，在阅读本节内容时，可以认为它是一个通用建议，最后您都会明白如何用它。我准备解释的就是示范如何前进的路线图。这里集中关注那些通用思想，然后在碰到疑问时，您可以返回来再次阅读本节内容而解除疑问。

那么，如果您有一个问题，并且希望使用过滤器和管道线解决这个问题，那么您该怎么办呢？下面是具体的解决步骤。

(1) 分解问题

首先是思考。找一个安静的地方，闭上眼睛，思考如何将问题分解成多个部分，每个部分都可以由一个单独的程序执行。此时，不用知道各部分任务需要使用什么工具来执行。您需要的就是思考。

当有经验的 Unix 人士思考问题时，他们不停地反复思考，从不同的视角观看问题，直至找到某些看上去可行的方法。然后他们查找完成任务的工具。接着进行试验，查看这种方法是否可行。如果您观察有经验 Unix 人士的工作，那么您会注意到他们永远不灰心丧气(好好地想一想)。

(2) 选择工具

Unix 程序有数百个之多，其中许多程序是过滤器，因此，为了熟练地使用 Unix，需要知道哪个程序最适合解决哪种类型的问题。当然，这听起来不大可能。您如何记住数百个程序的功能呢(更不用说细节了)?

实际上，您将会发现大多数 Unix 问题都可以从一个拥有大约 30 个程序的相当小的工具箱中选择过滤器来加以解决。多年以来，实践证明这些程序是那些最通用、最有用的程序，而这些程序就是我们在接下来的 4 章中要讨论的程序。出于参考目的，图 16-1(本章后面)中列举了这些重要的过滤器。

(3) 与他人讨论

一旦考虑了如何分解问题，并且选定了使用的工具，就可以找一些其他人，询问他们的建议。虽说在请求帮助之前最好先阅读一下手册(参见第 9 章)，但是传统上讲，Unix 一直是通过口授传授的。为了成为一名成熟的 Unix 人士，必须了解更多其他有经验的人如何解决问题。

(4) 选择选项

一旦研究了问题并选择了工具，就应该查看联机手册中的文档资料(参见第 9 章)。每个希望使用的程序都要查看联机手册。您的目标就是检查选项，查找与工作相关的选项。

大多数情况下，可以安全地忽略许多选项。但是，最好还是浏览一遍各个选项的描述，从而防止错过适合解决该问题的选项。否则，您可能面临做许多额外工作的风险，因为您不知道一个特定的选项是否可用。据我所知，最聪明、最有知识的人每天都要查看手册好几次。

提示

当使用重定向、过滤器和管道线解决问题时，3 个最重要的技能就是思考、RTFM*以及询问他人的意见。

16.5 可能最简单的过滤器：cat

过滤器每次一行地从标准输入读取数据，完成处理后，将结果每次一行地写到标准输出。那么哪个过滤器是最简单的过滤器呢？不做任何事的过滤器就是最简单的过滤器。

这个过滤器的名称就是 **cat**(稍后解释原因)，它所做的事情就是将标准输入的数据复制到标准输出，并且不以任何方式对数据做任何特殊处理或者改变。

下面是一个简单的例子，自己可以试一下。输入命令：

```
cat
```

运行该命令后，**cat** 程序将启动，并等待标准输入的数据。因为默认情况下，标准输入是键盘，所以 **cat** 命令等待键入内容。

键入一些自己希望的内容。在每行的末尾按下<Return>键。每次按下<Return>键时，刚才键入的那一行字符就发送给 **cat**，而 **cat** 将把它复制到标准输出，默认情况下就是屏幕。最终结果是每键入一行内容，该内容就在屏幕上显示两次，其中一次是您键入时显示的，一次是由 **cat** 命令显示的。例如：

```
this is line 1
this is line 1
this is line 2
this is line 2
```

输入结束后，按[^]D(<Ctrl-D>)键，即发送 **eof** 信号的键。这样就告诉 Unix 没有数据输入了(参见第 7 章)。**cat** 命令结束，又返回到 shell 提示符处。

现在，您可能要问，一个过滤器根本不做什么事还有什么用呢？实际上，该命令有几种应用，而且还相当重要。

因为 **cat** 不做任何事情，所以在管道线中使用它没有用(好好地想一想，直到理解了它的含义)。但是，当与 I/O 重定向组合使用时，**cat** 命令十分方便。当需要在管道线的开头组合不止一个文件时，**cat** 命令也十分有用。下面举一些例子。

cat 命令的第一种应用就是与重定向组合，快速地创建一个小文件。考虑下面的命令：

```
cat > data
```

标准输入(默认情况下)是键盘，但是标准输出被重定向到文件 **data**。因此，键入的每行在按下<Return>键时被直接复制到这个文件中。您可以输入任意多行数据，并在结束时按下[^]D 键告诉 **cat** 不再有数据了。

* RTFM 的含义在术语表和第 9 章中有解释。

如果文件 **data** 还不存在, 那么 Unix 就会创建这个文件。如果已经存在, 那么这个文件的内容将被替换。通过这种方式, 可以使用 **cat** 命令创建新文件, 或者替换已有文件的内容。在需要创建或者取代一个小文件时, 有经验的用户经常这样做。

说“小文件”的原因在于当按<Return>键时, 刚键入的那一行才复制到标准输出。如果希望改变这一行, 则必须停止 **cat**, 然后重新启动 **cat**, 再重新键入所有的内容。

我发现通过这种方式快速地创建或者取代一个小文件(假如说最多有 4 到 5 行)是一种有效的方法。使用 **cat** 要比启动文本编辑器(例如 **vi** 或者 **Emacs**)键入文本, 然后再停止文本编辑器更快(而且更有趣)。当然, 这种方式很容易犯错, 如果我希望键入的行数超过 5 行, 那么我将使用编辑器, 因为这样允许我对键入的内容进行修改。

cat 命令的第二种应用是在已有文件中追加少数几行内容。这样做时需要使用 **>>** 重定向标准输出, 例如:

```
cat >> data
```

现在, 无论键入什么内容都会追加到文件 **data** 中(正如第 15 章中的解释, 当使用 **>>** 重定向输出时, **shell** 追加输出)。

cat 命令的第三种应用是显示一个短文件。只需将标准输入重定向到希望显示的文件即可。例如:

```
cat < data
```

在这个例子中, 输入来源于文件 **data**, 输出显示在屏幕上(默认情况)。换句话说, 就是显示文件 **data** 的内容。当然, 如果这个文件的行数比屏幕能够显示的行数大, 那么有一些行将在阅读之前滚动出屏幕的范围。在这种情况下, 不能使用 **cat** 显示文件, 而应该使用 **less**(参见第 21 章)每次一屏地显示文件。

cat 命令的第四种应用就是显示任何文件的最后一部分。假设您使用前面的命令显示文件 **data**。如果 **data** 是一个短文件, 可以在屏幕上完全显示, 那么一切正常, 没有什么问题。但是, 如果 **data** 文件的行数比屏幕能够显示的行数大, 那么除了最后一部分, 其他部分都将滚动出屏幕的范围。通常, 这发生得很快, 剩下的就是文件的最后一部分, 其大小为满屏幕, 而这正是我们希望的。

如果希望自己试一下这个命令, 那么可以用 **cat** 命令显示第 6 章中讨论的一个配置文件。例如, 试一试下述命令:

```
cat < /etc/profile
```

注意, 当显示长文件时, **cat** 是如何只让用户查看最后一部分的。

为了方便起见, 如果省略了 **<** 字符, **cat** 将直接从文件中读取数据(我们将在下一节中讨论)。因此, 如果希望体验一下第 6 章中的配置文件, 那么使用的命令如下所示:

```
cat /boot/grub/menu.lst
cat /etc/hosts
cat /etc/inittab
cat /etc/passwd
cat /etc/profile
```



```
cat /etc/samba/smb.conf
```

注意：(1)如果使用的不是 Linux，那么系统上可能没有所有这些文件。(2)在大多数系统上，显示 **menu.lst** 文件需要超级用户的权限。

在体验这些命令时，您将会发现，如果文件是短的，那么 **cat** 命令将快速地展示该文件。如果文件是长的，那么文件的大部分将快速地滚动出屏幕，从而使您无法阅读文件。正如我们所讨论的，您所看到的就是文件的最后一部分(显示满屏幕)。

在本章后面，我们还会讨论另一个程序 **tail**，这个程序可以用来快速地显示文件的末尾。大多数时候，**tail** 命令要比 **cat** 命令出色。但是，在一些情况中，**cat** 命令实际上是最佳选择。这是因为 **tail** 只显示指定的行数，默认值是 10；而 **cat** 命令将显示满屏幕。为了明白我的意思，请对上面列举的文件使用 **tail** 命令，看看出现什么情况。例如：

```
tail /etc/profile
```

cat 命令的第五种应用是通过重定向标准输入和输出复制文件。例如，为了将文件 **data** 复制到另一个文件 **newdata** 中，可以输入：

```
cat < data > newdata
```

当然，Unix 还拥有更好的命令来复制文件。这个命令就是 **cp**，我们将在第 25 章中讨论这个命令。但是，有趣的是，当需要时使用 **cat** 命令也可以复制文件。

cat 命令还有更多的应用，这些我们将在下一节中介绍。在这之前，我们先思考一些真正不平凡的东西。我们从可能最简单的过滤器 **cat** 入手，根据定义，该过滤器不做任何事情。但是，通过将 **cat** 与 I/O 重定向一起使用，我们能够使它昂起头来，执行许多技巧(本章后面的图 16-2 对此进行了汇总)。

对 **cat** 能力的考察为我们提供了一个很好的例子，它体现了 Unix 设计的精美之处。一个看起来非常简单的概念——数据从标准输入流向标准输出，却会产生这么多意想不到的结果。看看我们用一个不做任何事的过滤器做了多少事情，就可以想一想还会有什么事情做不出来！

提示

Unix 的精巧之处部分体现在您会突然一下子恍然大悟，自言自语地说：“噢，这就是他们要这样做的原因。”

16.6 增强过滤器的功能

通过对过滤器做一点有效的改进，就有可能使其功能大增。这点改进就是允许指定一个或多个输入文件名。

众所周知，过滤器的严格定义要求它从标准输入读取数据。如果希望过滤器从文件读取数据，则必须将标准输入重定向到这个文件，例如：

```
cat < data
```

但是，如果将要从中读取数据的文件的名称指定为一个参数会怎么样呢？例如：

```
cat data
```

这实际上就是 **cat** 命令的情况，而且上面两条命令是相等的。因此，为了快速地显示一个短文件，只需键入 **cat** 命令，后面跟一个文件的名称即可，例如上面最后一个例子(对于较长的文件，可以使用 **less**，参见第 21 章)。

起初，这一点小小的改变(省略<字符)似乎没有什么意义，但是情况并非如此。的确，我们使命令行稍微简单了一点，这样将方便一点，但是这是有代价的。**cat** 程序本身必定变得更复杂。它不但必须能从标准输入读取数据，而且还必须能从任意文件中读取数据。此外，通过扩展 **cat** 的功能，我们就失去了纯过滤器所具有的优美和简洁。

然而，许多过滤器正是按这种方式扩展的，这不仅是因为它使得从一个文件读取数据变得简单了，而且还因为它使得从多个文件中读取数据成为可能。例如，下面就是 **cat** 命令语法的缩写版本：

```
cat [file...]
```

其中 *file* 是文件的名称，过滤器从这个文件中读取数据。

注意 *file* 参数之后的 3 个圆点。这意味着可以指定不止一个文件名(有关命令语法的解释请参见第 10 章)。

因此，这里有一点很重要，即在扩展 **cat** 能够从文件中读取数据的功能时，我们同时允许它从不只一个文件中读取数据。当为输入指定多个文件时，**cat** 将轮流从每个文件中读取所有的数据。在读取数据的过程中，**cat** 按它遇到的顺序将文本的每一行写入到标准输出。这意味着我们可以使用 **cat** 将任意多个文件的内容组合在一起。

这是一个非常重要的概念，因此要花一点时间仔细地考虑一下下面的例子：

```
cat name address phone
cat name address phone > info
cat name address phone | sort
```

第一个例子将多个文件(**name**、**address** 和 **phone**)的内容组合在一起，并显示在屏幕上；第二个例子组合相同的文件，并将数据写到另一个文件(**info**)中；第三个例子将数据管道传递给一个程序(**sort**)做进一步处理。

正如前面所述，不止是 **cat**，许多其他过滤器也可以从多个文件读取输入。从技术上讲，这并不必要。如果希望操作不止一个文件中的数据，那么我们可以使用 **cat** 收集数据，然后通过管道传送给希望的过滤器。例如，假设您希望组合这 3 个文件中的数据，然后对数据进行排序。**sort** 命令不必从多个文件中读取数据。我们所需做的就是使用 **cat** 将数据组合起来，然后管道传送给 **sort**：

```
cat name address phone | sort
```

这在某种意义上有很大的吸引力。通过将 **cat** 扩展成可以从多个文件中读取数据，而不仅仅是从标准输入读取数据，我们已经失去了 Unix 总体设计的某些精美性。但是，通过使用 **cat** 为其他过滤器服务，至少可以保持其他过滤器的纯洁性。

就如生活的众多方面一样，实用性战胜了美丽和纯洁性。但是，在需要将文件的组合发送给过滤器时，每次都要使用 **cat** 也是很麻烦的。所以，大多数过滤器允许指定多个文件名作为参数。

例如，下面的3条命令都排序不止一个文件中的数据。第一条命令在屏幕上显示输出；第二条命令将输出保存到文件中；第三条命令将输出管道传送给另一个程序做进一步处理（现在还不用关心命令的细节）。

```
sort name address phone
sort name address phone > info
sort name address phone | grep Harley
```

此时，我希望考虑下述哲理问题。根据定义，过滤器必须从标准输入读取数据。这是否意味着，如果一个程序从文件中读取输入数据，那么这个程序就不是一个真正的过滤器了呢？

答案有两种可能，两种答案都可以接受。第一，我们可以认为当一个像 **cat** 或 **sort** 这样的程序从标准输入读取数据时，它充当的是过滤器；但是当它从文件中读取数据时，它就不是过滤器了。这种方法维护了系统的纯洁性。但它意味着大量的程序可能是也可能不是过滤器，这取决于它们的使用方式。

第二，我们可以扩展过滤器的定义，即无论从标准输入还是从文件中读取数据，它都是过滤器。该定义比较实际，但是它以牺牲原始设计的一些优美为代价。

16.7 最有用的过滤器列表

此时，我们已经讨论了与过滤器相关的基本思想。在本章的剩余部分以及第17、18和19章中，我们将逐个讨论各种不同的过滤器。作为概括，图16-1列举了我认为最有用的 Unix 过滤器。

过滤器	章号	参阅	作用
awk	—	perl	编程语言：操作文本
cat	16	split 、 tac 、 rev	组合文件；复制标准输入到标准输出
colrm	16	cut 、 join 、 paste	删除指定的数据列
comm	17	cmp 、 diff 、 sdiff	比较两个有序文件，显示区别
cmp	17	comm 、 diff 、 sdiff	比较两个文件
cut	17	colrm 、 join 、 paste	从数据中抽取指定列(字段)
diff	17	cmp 、 comm 、 sdiff	比较两个文件，显示不同
expand	18	unexpand	将制表符转变为空格
fold	18	fmt 、 pr	将长行格式化成较短的行
fmt	18	fold 、 pr	格式化段落，从而使它们看上去更漂亮
grep	19	look 、 strings	选择包含指定模式的行
head	16	tail	从数据的开头选择行

图 16-1 最有用的 Unix 过滤器

join	19	colrm、cut、paste	基于公用字段，组合数据列
look	19	grep	选择以指定模式开头的行
nl	18	wc	创建行号
paste	17	colrm、cut、join	组合数据列
perl	—	awk	编程语言：操作文本、文件、进程
pr	18	fold、fmt	将文本格式化为页或者列
rev	16	cat、tac	每行数据中的字符反序排列
sdiff	17	cmp、comm、diff	比较两个文件，显示区别
sed	19	tr	非交互式文本编辑
sort	19	tsort、uniq	排序数据：检查数据是否有序
split	16	cat	将大文件分隔成较小的文件
strings	19	grep	在二进制文件中搜索字符串
tac	16	cat、rev	组合文件，同时将文本行的顺序反转
tail	16	head	从数据的末尾选择行
tr	19	sed	改变或者删除选定的字符
tsort	19	sort	根据偏序创建全序
unexpand	18	expand	将空格转变成制表符
uniq	19	sort	选择重复/唯一行
wc	18	nl	统计行数、单词数和字符数

图 16-1 (续)

本表示范了最重要的 Unix 过滤器，其中大多数过滤器都是 30 年前的。使用该列表中的过滤器可以解决许多不同类型的问题。大多数情况下，只需要使用一个过滤器；使用的过滤器极少超过 4 个。

awk 和 **perl** 是复杂的编程语言，可以用来编写在管道线中担当过滤器的程序。更多的相关信息，可以先查找联机手册(**man awk**, **man perl**)，然后再在网络上搜索，在网络上可以查找到大量的信息。

不管正在使用的系统是 Unix 还是 Linux，您都会发现在这 4 章中阅读的内容大多数都适合您的系统。这是因为我们将讨论的过滤器的基本细节在各种系统中都是相同的。实际上，这些过滤器大多数 30 年来都一直以相同的方式运转。

在继续之前，我先提醒一下，无论何时，都应该使用 **man** 命令显示程序的说明书页，查看程序在自己系统上的运转方式的权威指南。如果您使用的是 GNU 实用工具——Linux 和 FreeBSD 使用的就是 GNU 实用工具(参见第 2 章)，那么您也可以使用 **info** 命令访问 Info 系统。例如：

```
man cat
info cat
```

对于大多数 GNU 实用工具来说，使用 **--help** 选项可以显示其命令的语法和选项摘要。例如：

```
cat --help
```

有关 **man** 和 **info** 命令使用方式的讨论，请参见第 9 章。有关语法的讨论，请参见第 10 章。

16.8 组合文件：cat

相关过滤器：**rev**、**split**、**tac**

cat 程序将数据未加改变地复制到标准输出。数据可以来源于标准输入或者一个或多个文件。**cat** 命令的语法为：

```
cat [-bns] [file...]
```

其中 *file* 是文件的名称。

我们已经讨论了 **cat** 程序的几种使用方式。但是，到目前为止，**cat** 程序的最重要应用就是组合多个文件。下面举一些典型的例子，这些例子将 3 个文件组合在一起。当然，使用的文件数量可以任意多。

```
cat name address phone
cat name address phone > info
cat name address phone | sort
```

这些模式值得记住，因为今后会大量使用它们。出于参考目的，图 16-2 对这些模式进行了总结。

语法	作用
cat > file	从键盘读取数据，创建新文件或者替换已有文件
cat >> file	从键盘读取数据，将数据追加到已有文件中
cat < file	显示一个已有文件
cat file	显示一个已有文件
cat < file1 > file2	复制文件
cat file1 file2 file3 less	组合多个文件，每次一屏地显示结果
cat file1 file2 file3 > file4	组合多个文件，将输出保存到一个不同文件中
cat file1 file2 file3 program	组合多个文件，将输出管道传送给另一个程序

图 16-2 cat 程序的多种应用

cat 程序是最简单的过滤器。它从标准输入读取数据，并将数据不加修改地写入到标准输出。尽管如此简单，但 **cat** 程序可以执行的任务数量惊人，本表总结了这些任务。这样一个简单过滤器的强大功能来源于丰富的 Unix I/O 重定向和管道线能力。

大多数时候，**cat** 用于组合文件，或者进行显示(将输出管道传送给 **less**)，或者保存在另一个文件中(将标准输出重定向到文件)，或者管道传送给另一个程序做进一步处理。详情请参见正文。

在第一个例子中，**cat** 读取并组合 3 个文件(这 3 个文件分别是 **name**、**address** 和 **phone**)

的内容，并将输出显示在屏幕上。通常情况下，只有在这些文件非常短，从而使组合输出不会滚动出屏幕范围时，才使用这样的命令。极有可能的是需要将输出管道传送给 **less**(参见第 21 章)，从而每次一屏地显示输出。例如：

```
cat name address phone | less
```

第二个例子还是组合这 3 个文件，但是将标准输出重定向到另一个文件(在这个例子中是 **info**)。如果这个文件不存在，shell 将创建这个文件。如果这个文件存在，那么这个文件将被替换，这意味着该文件中的原始数据将永远丢失(有关重定向和文件替换的讨论，请参见第 15 章)。

第三个例子也组合相同的文件，并将输出管道传送给另一个程序做进一步处理，在这个例子中，这个程序就是 **sort**(参见第 19 章)。

当使用 **cat** 程序进行组合时，有一个常见的错误必须避免：不要将输出重定向到一个输入文件上。例如，假设您希望将文件 **address** 和 **phone** 的内容追加到文件 **name** 中。您可能会想，只需组合这 3 个文件，并将结果保存在 **name** 中即可：

```
cat name address phone > name
```

该命令无法完成任务，原因在于 shell 处理重定向的方式。在程序将标准输出重定向到文件之前，shell 必须确保这个文件存在并且是空的。在这个例子中，如果 **name** 文件不存在，shell 将创建这个文件。如果 **name** 文件已经存在，shell 将清空这个文件。在我们的例子中，到 **cat** 准备好读取 **name** 文件的内容时，这个文件早已被清空。

当输入一条类似于上面的命令时，将看到类似于下面的消息：

```
cat: name: input file is output file
```

这看上去像一个警告消息，但是实际上，已经为时太晚了。即便是按下 **^C** 键(终止程序)也没有什么作用了。当看到这个消息时，**name** 文件的内容早已被删除。

将 **address** 和 **phone** 文件中的内容追加到文件 **name** 中的安全方式是使用：

```
cat address phone >> name
```

注意，我们没有使用输出文件作为输入文件。更准确地说，就是我们将所有其他文件中的内容追加到输出文件中。

下面总结 **cat** 的讨论，最有用的选项如下所示：

- **-n**(number, 数字)选项在每行前面加一个行号。
- **-b**(blank, 空白)选项和 **-n** 选项一起使用，告诉 **cat** 不要对空白行编号。
- **-s**(squeeze, 挤压)选项将多个连续空白行替换为一个空白行。

名称含义

cat

cat 程序的主要应用是将多个文件的内容组合成一个单独的输出流。基于这一原因，很自然地会假定 **cat** 代表“concatenate, 连接”。实际上，情况并非如此。

cat 的名称来源于一个古老的单词“catenate”，它意味着“to join in a chain”。所有受过一流教育的 Unix 用户都知道，catena 就是 chain 的拉丁语单词。

16.9 划分文件：split

相关过滤器：**cat**

我们刚刚讨论了如何使用 **cat** 将两个或者更多个文件组合成一个大的文件。那么这一过程的逆过程：将一个大文件划分成几个较小的文件，又该怎么办呢？可以使用 **split** 程序。该程序的语法为：

```
split [-d] [-a num] [-l lines] [file [prefix]]
```

其中，*num* 是创建文件名时用作文件名后缀的字符或数字数量；*lines* 是每个新文件所包含行的最大数量；*file* 是输入文件的名称；*prefix* 是创建文件时使用的名称。

split 程序开发于 20 世纪 70 年代初，当时大的文本文件可能会产生问题。那个时代，硬盘存储空间受限，而且处理器很慢^{*}。现在，大型的硬盘已经普及，而且计算机也相当快。结果是，在存储和操作大型文本文件时，我们极少遇到问题。然而，有时候，仍会希望将一个大的文件划分成若干部分，此时使用 **split** 可以节省大量的时间。例如，您可能希望向某些人发送一个非常大的文件，而这个电子邮件账户在接受的消息大小上有限制。

默认情况下，**split** 创建 1000 行长的文件。例如，假设您拥有一个文件 **data**，这个文件共有 57984 行，您希望将它分解成较小的文件。您使用了命令：

```
split data
```

这将创建 58 个新文件：57 个包含有 1000 行数据的文件，以及最后一个包含剩余的 984 行数据的文件。

如果希望改变这些文件的最大大小，可以使用 **-l**(lines, 行)选项。例如，如果将 **data** 文件(有 57 984 行)分隔成包含有 5000 行数据的文件，可以使用：

```
split -l 5000 data
```

该命令将创建 12 个文件：11 个包含有 5000 行数据的文件(总共 55 000 行)，和最后一个包含有 2984 行(余数)数据的文件。

提示

当使用要求大数字的选项时，不要键入逗号(或者点号——欧洲标准)将数字分组。例如，应该使用：

```
split -l 5000 data
```

而不应该使用：

```
split -l 5,000 data
```

如果使用逗号，会产生一个语法错误，得到一个类似于下面的错误消息：

```
split: 5,000: invalid number of lines
```

^{*}1976 年，当我还是一年级研究生时，我使用 Unix 编写一个算术操作文件中数据的 C 程序，以现在的标准来看，这个程序相对较小。但在那时候，这个文件已经不小了，而且计算机没有足够的内存装载所有的数据。因此，我的程序不得不非常复杂，以便能够一小块一小块地处理数据(需要时数据就交换进内存，不需要时就交换出内存)。

现在您可能奇怪,所有这些新文件的名称是什么呢?如果 **split** 自动创建文件,那么这些文件应该使用合理的名称。此外,必须注意不要由于不小心而替换了已有的文件。

默认情况下, **split** 使用以字母 **x** 开头的名称,后面跟两个字符的后缀。后缀是 **aa**、**ab**、**ac**、**ad** 等。例如,在上一个例子中, **split** 创建了 12 个新文件,这些文件名(默认)为:

```
xaa xab xac xad xae xaf xag xah xai xaj xak xal
```

如果 **split** 生成的文件多于 26 个,那么文件名 **xaz** 之后就是 **xba**、**xbb**、**xbc**, 依此类推。因为字母表中只有 26 个字母,所以总共允许 676(26×26) 个新文件名,即从 **xaa** 到 **xzz**。

如果不喜欢这些文件名,则有两种方法改变它们。第一种,如果使用了 **-d**(digits, 数字)选项,那么 **split** 就在文件名后面使用两个数字后缀(从 **00** 开始),而不是两个字母后缀。例如,下述命令使用我们在上一个例子中使用的同一个文件 **data**(包含 57 984 行数据):

```
split -d -l 5000 data
```

12 个新文件被分别命名为:

```
x00 x01 x02 x03 x04 x05 x06 x07 x08 x09 x10 x11
```

如果不希望文件名以 **x** 开头,可以指定自己的名称用作所生成文件名的前缀,例如:

```
split -d -l 5000 data harley
```

新文件被命名为:

```
harley00 harley01 harley02 harley03 harley04 harley05  
harley06 harley07 harley08 harley09 harley10 harley11
```

当对 **split** 命令使用 **-d** 选项时,可以创建 100 个文件(10×10),使用的后缀从 **00** 到 **99**。如果没有 **-d** 选项,则可以创建多达 676 个文件(26×26),使用的后缀从 **aa** 到 **zz**。如果需要更多的文件,可以使用 **-a** 选项,后面给出希望在后缀中使用的数字或者字符的数量。例如:

```
split -d -a 3 data
```

新的文件名将使用 3 位数字后缀:

```
x000 x001 x002 x003...
```

同样,也可以在没有 **-d** 选项的情况下使用 **-a** 选项:

```
split -a 3 data
```

在这种情况下,新的文件名将使用 3 个字母后缀:

```
xaaa xaab xaac xaad...
```

通过这种方式,可以使用 **split** 命令分解大的输入文件,而不必担心用完文件名。

默认情况下, **split** 创建包含 1000 行数据的文件。但是,正如前面所述,可以创建任意大小的文件,甚至是更小的文件。下面举一个典型的日常应用例子,我确信您熟悉这种情况。

您在为一位有实力的美国参议员工作,他正在竞选美国总统。离选举日还有两周的时候,竞选活动遇到了困难。参议员很失望,因为您熟悉 Unix,所以他提名您为新的竞选主管。

您第一天的工作就是处理一个很大的文本文件,文件名是 **supporters**, 在这个文件中,每行都包含一个潜在投票人的姓名和电话号码。您的工作就是组织全国各地的志愿者联系列表上的人,劝说他们为您的候选人投票。您认为剩下的时间每名志愿者只能联系 40 人。

所以您登录 Unix 系统，输入命令：

```
split -d -l 40 supporters voter
```

现在您有一组文件，分别命名为 **voter00**、**voter01**、**voter02** 等。每个文件(可能除了最后一个文件)都包含有 40 个姓名。您命令您的员工通过电子邮件给每名志愿者发送一个文件，并指示他们联系列表中的 40 个投票人。

因为您的勤奋工作以及高超的 Unix 技能，您的候选人当选了美国总统。一周之后，您被任命了一个更有权力更有影响力的职位。

16.10 组合文件时反转文本行的顺序：tac

相关过滤器：**cat**、**rev**

正如前面讨论的，**cat** 是所有过滤器中最基本的过滤器，也是最有用的一个过滤器。**tac** 程序与 **cat** 相似，但它们之间有一个主要的区别，即 **tac** 在将文本写入到标准输出之前将文本行的顺序反转(名称 **tac** 就是 **cat** 的反向拼写)。

tac 的语法为：

```
tac [file...]
```

像 **cat** 一样，**tac** 从标准输入读取数据，并将输出写入标准输出，而且 **tac** 也可以组合输入文件。例如，假设您有一个文件 **log**。您希望将文件 **log** 中各行的顺序反转，然后将结果写入一个新文件 **reverse-log** 中。所使用的命令为：

```
tac log > reverse-log
```

例如，假设 **log** 文件包含：

```
Oct 01: event 1 took place
Oct 02: event 2 took place
Oct 03: event 3 took place
Oct 04: event 4 took place
```

在运行了上面的 **tac** 命令之后，**reverse-log** 文件中将包含：

```
Oct 04: event 4 took place
Oct 03: event 3 took place
Oct 02: event 2 took place
Oct 01: event 1 took place
```

此时，您可能会觉得 **tac** 除了奇怪之外没有什么实际意义。或许人们编写它只是因为有人认为它的名称很有趣(**cat** 反过来)。但是，您曾经实际需要过这个程序吗？

答案是您不经常需要使用 **tac**。但是，当需要它时，它就珍贵无比。例如，假设您有一个程序，这个程序向日志文件中写注释(经常发生的情况)。最旧的注释位于日志文件的开头，最新的注释位于日志文件的末尾。这个文件命名为 **log**，现在已有 5000 行长，而您希望按照从新到旧的顺序显示注释。

如果没有 **tac** 程序，那么就没有以相反顺序显示长文件中各行的简单方法。有了 **tac**

程序之后，一切就简单了。只需使用 **tac** 命令将文件反序，并管道传送给 **less**(参见第 21 章)：

```
tac log | less
```

如果需要组合文件，**tac** 也可以完成，例如：

```
tac log1 log2 log3 | less
```

该命令将把 3 个文件分别反序，然后再组合它们，最后管道传送给 **less**。

16.11 反转字符的顺序：rev

相关过滤器：**cat**、**tac**

tac 程序将文件中各行的顺序反转，那么如果希望将各行中字符的顺序反转，该怎么办呢？在这种情况下，可以使用 **rev**。**rev** 的语法为：

```
rev [file...]
```

其中 *file* 是文件名。

下面举一个例子。假设您有一个文件 **data**，该文件包含：

```
12345
```

```
abcde
```

```
AxAxA
```

您输入：

```
rev data
```

输出为：

```
54321
```

```
edcba
```

```
AxAxA
```

假设您希望将每行中字符的顺序反转，并且将文件中行的顺序反转，则只需将 **rev** 的输出管道传送给 **tac** 即可，例如：

```
rev data | tac
```

输出为：

```
AxAxA
```

```
edcba
```

```
54321
```

请您自行考虑，如果先使用 **tac** 会发生什么情况呢？

```
tac data | rev
```

结束本节之前，我们考虑一个复杂的例子。假设您有一个文件 **pattern**，该文件包含下述内容：

```

X
XX
XXX
XXXX

```

考虑下述 4 条命令的输出(\$是 shell 提示符):

```

$cat pattern
X
XX
XXX
XXXX

```

```

$tac pattern
XXXX
XXX
XX
X

```

```

$rev pattern
X
XX
XXX
XXXX

```

```

$rev pattern | tac
XXXX
XXX
XX
X

```

对于上述命令，您都能理解吗？

16.12 从数据开头或末尾选择数据行：head、tail

当拥有的数据太多，不容易理解时，有两个程序可以用于快速地选取部分数据，这两个程序就是 **head** 和 **tail**。其中，**head** 从数据的开头选择数据行，而 **tail** 从数据的末尾选择数据行。

很多时候都需要使用 **head** 和 **tail** 显示文件的开头或末尾。基于这一原因，我将这两条命令的讨论延迟到第 21 章中，那一章将讨论文件显示命令。在本节中，我们将讨论如何在管道线中把 **head** 和 **tail** 作为过滤器使用。

当把 **head** 和 **tail** 作为过滤器使用时，语法比较简单：

```

head [-n lines]
tail [-n lines]

```

其中 *lines* 是希望选取的数据行的数量(在第 21 章，我们将使用更复杂的语法)。

默认情况下，**head** 和 **tail** 都选取 10 行数据。例如，假设您有一个程序 **calculate**，该程序生成许多数据。为了显示前 10 行数据，可以使用：

```
calculate | head
```

为了显示最后 10 行数据，可以使用：

```
calculate | tail
```

如果希望选取不同数量的数据行，则可以使用一个连字符(-)或者 -n 选项，后面跟上这个数量。例如，为了选取 15 行数据，可以使用：

```
calculate | head -n 15
calculate | tail -n 15
```

通常在复杂管道线的末尾使用 **head** 和 **tail**，以选取由前面命令生成的部分数据。例如，假设您有 4 个文件：**data1**、**data2**、**data3** 和 **data4**。您希望将这 4 个文件的内容组合起来，进行排序后，显示前 20 行和最后 20 行数据。

为了组合这些文件，可以使用 **cat** 程序(本章前面讨论过)。为了执行排序，可以使用 **sort** 程序(第 19 章)：

```
cat data1 data2 data3 data4 | sort | head -n 20
cat data1 data2 data3 data4 | sort | tail -n 20
```

有时候，希望将 **head** 或 **tail** 的输出发送给另一个过滤器。例如，在下述管道线中，我们使用 **head** 从 **sort** 输出的开头选取 300 行数据，然后发送给 **less**(参见第 21 章)每次一屏地进行显示：

```
cat data1 data2 data3 data4 | sort | head -n 300 | less
```

同样，将 **head** 或 **tail** 的输出保存到文件中通常比较方便。下述例子从输出中选取最后 10 行数据，并保存到 **most-recent** 文件中：

```
cat data1 data2 data3 data4 | sort | tail > most-recent
```

提示

最初，**head** 和 **tail** 不要求使用 -n 选项，只需键入一个连字符后面跟一个数字即可。例如，下述命令都显示 15 行输出：

```
calculate | head -n 15
calculate | tail -n 15
calculate | head -15
calculate | tail -15
```

正式地讲，现代版本的 **head** 和 **tail** 理应需要 -n 选项，这就是为什么我包含它的原因。但是，大多数版本的 Unix 和 Linux 都接受两种类型的语法，因此您通常可以省略 -n 选项。

16.13 删除数据列：colrm

相关过滤器：**cut**、**paste**

colrm(“column remove”，列移除)程序从标准输入读取数据，删除指定的数据列，然后将剩余数据写入标准输出。**colrm**的语法为：

```
colrm [startcol [endcol]]
```

其中 *startcol* 和 *endcol* 指定要移除区域的开头和末尾。列的编号从 1 开始。

下面举例说明：您是加利福尼亚某大学的一名终身教授，您需要 PE 201 班(Intermediate Surfing，中级冲浪)中所有学生的成绩表。该成绩表不需要显示学生的姓名。

您有一个主数据文件，即 **students**，这个文件中为每名学生包含了一行信息。每行信息有学号、姓名、期末考试成绩和课程成绩：

```
012-34-5678  Ambercrombie, Al  95%  A
123-45-6789  Barton, Barbara  65%  C
234-56-7890  Canby, Charles   77%  B
345-67-8901  Danfield, Deann  82%  B
```

为了构建这张成绩表，需要移除姓名，即列 14 至列 30(包含这两列)。使用下述命令：

```
colrm 14 30 < students
```

输出为：

```
012-34-5678 95% A
123-45-6789 65% C
234-56-7890 77% B
345-67-8901 82% B
```

作为对管道和重定向技术的回顾，下面再示范两个例子。第一个例子，如果成绩表很长，则可以将成绩表管道传送给 **less**，每次一屏地显示数据：

```
colrm 14 30 < students | less
```

第二个例子，如果希望保存输出，则可以将输出重定向到文件：

```
colrm 14 30 < students > grades
```

如果只指定了起始列，那么 **colrm** 将移除从该列开始到这一行末尾的所有列。例如：

```
colrm 14 < students
```

将显示：

```
012-34-5678
123-45-6789
234-56-7890
345-67-8901
```

如果既没有指定起始列也没有指定结束列，那么 **colrm** 不删除任何列。

16.14 练习

1. 复习题

1. 什么是过滤器？为什么过滤器如此重要？
2. 假设您需要使用过滤器和管道线解决一个困难的问题。那么您需要遵循哪 4 个步骤？您需要掌握哪 3 种最重要的技能？
3. 为什么 **cat** 是最简单的过滤器？尽管很简单，但 **cat** 可以用于许多用途。请列举 4 种用途。
4. **tac** 和 **rev** 之间有什么区别？

2. 应用题

1. 一名科学家做了一个实验，生成的数据存储在文件 **data1**、**data2**、**data3**、**data4** 和 **data5** 中。他希望知道这些数据总共有多少行。命令 **wc -l** 从标准输入读取数据，并统计行的数量。如何使用这个命令统计 5 个文件中行的总数量？
2. 有一个文本文件 **important**，那么使用哪条命令可以按照下述 4 种不同方式显示这个文本文件的内容：(a)显示该文件的原始内容；(b)将行反序显示；(c)将每行中的字符反序显示；(d)将行和字符都反序显示；对于(b)、(c)和(d)方式，哪条命令可以执行相反的转换？如何进行测试？
3. 在第 6 章中，我们讨论了 Linux 程序 **dmesg**，该程序显示系统启动过程中生成的消息。一般情况下，这样的消息太多了。那么，使用什么命令来显示最后 25 条启动消息呢？

3. 思考题

1. 图 16-1 列举了最重要的 Unix 过滤器。不算 **awk** 和 **perl** 过滤器(它们是编程语言)，列表中总共有 19 个不同的过滤器。对于大多数问题来说，只需要使用一个过滤器；另外，所需的过滤器数量极少会超过 4 个。您论文为什么会这样呢？依您的观点来看，您认为还有哪些工具这个列表没有列举出来呢？
2. **split** 程序开发于 20 世纪 70 年代初，那时大的文本文件可能产生问题，因为磁盘是相对慢速的存储器，并且十分昂贵。现在，磁盘已经很快，而且还很便宜。那么我们还需要像 **split** 一样的程序吗？为什么呢？

过滤器：比较和抽取

在第 16 章中，我们花费了大量的时间讨论过滤器：过滤器就是能够从标准输入读取文本数据并向标准输出写入文本数据(每次一行)的程序。通过我们的讨论所得到的结论之一，就是过滤器通常被设计成工具，并且出色地完成一件事件。在接下来的 3 章中，我们将讨论许多具体的过滤器，在您阅读并思考各个例子时，您会发现该结论特别重要。

在本章中，我们将讨论那些用来比较文件以及从文件中抽取部分内容的过滤器。刚开始时，您可能认为这些话题都是些没有意思的话题，这不怪您。不过，本章将提供足够的细节——足以喂饱一匹大型马。在阅读这些细节，并开始明白隐藏在这些过滤器设计之后的聪明才智时，您会体会到它们实际上是多么得有趣。实际上，本章将讨论的过滤器不仅有趣，而且还是 Unix 工具箱中最有用和最重要的程序。

17.1 比较文件

多年以来，Unix 程序员创建了大量的工具来帮助回答下述问题：两个文件是否包含完全相同的数据？如果不相同，那么这两个文件之间有什么区别呢？比较两个文件要比想象的更复杂，因为您可以采取各种各样的方式来进行比较及显示结果。

在接下来的几节中，我们将讨论这类工具中最重要的工具。具体而言，我们将介绍这些工具能干什么，比较什么类型的文件以及哪些选项最有用。在这个过程中，我们还将示范一些例子，并提供一些重要的提示。我的目标比较简单：无论何时，当需要比较文件时，您能够快速分析情形，决定使用哪个程序和哪些选项，并且能够解释结果。

图 17-1 中汇总了最重要的文件比较程序(后面将讨论这些程序)。出于完整性考虑，该图中还包含了与排序文件以及从文件中选取数据相关的程序。我们将在第 19 章中讨论这些程序。

过滤器	作用	章号	文件类型	文件数量
cmp	比较两个文件	17	二进制或者文本	2 个
comm	比较两个有序文件, 显示区别	17	文本: 有序	2 个
diff	比较两个文件, 显示区别	17	文本	2 个
sdiff	比较两个文件, 显示区别	17	文本	2 个
cut	从数据中抽取指定列(字段)	17	文本	1 个或多个
paste	组合数据列	17	文本	1 个或多个
sort	排序数据	19	文本	1 个或多个
uniq	选取重复/唯一行	19	文本: 有序	1 个
grep	选取包含指定模式的行	19	文本	1 个或多个
look	选取以指定模式开头的行	19	文本: 有序	1 个

图 17-1 比较、排序及从文件中选取数据的程序

Unix 中有许多比较文件的程序, 最重要的程序有 **comm**、**cmp**、**diff** 和 **sdiff**(可以认为是 **diff** 的一种变体)。与此紧密相关的程序有排序文件(**sort**)、从文本中选取行(**uniq**)以及从文件中抽取部分内容(**cut**、**grep**、**look**)的程序。

本表汇总了这些程序, 说明了它们使用的数据类型(二进制或文本, 有序或无序)、使用文件的数量以及它们的主要作用。详情请参见正文(排序和选择程序在第 19 章中讨论。二进制文件和文本文件在第 19 章和第 23 章中讨论)。

17.2 比较任意两个文件: **cmp**

相关过滤器: **comm**、**diff**、**sdiff**

cmp 的使用只有一种情形: 查看两个文件是否相同。**cmp** 程序的语法为:

```
cmp file1 file2
```

其中 *file1* 和 *file2* 是文件的名称。

cmp 程序逐字节地比较两个文件, 查看两个文件是否相同。如果两个文件中对应的字节完全相同, 那么这两个文件就是相同的, 在这种情况下, **cmp** 程序不做任何处理(没有消息就是好消息)。如果两个文件不相同, 那么 **cmp** 程序就显示一个适合的消息。

例如, 假设您有一个程序拥有两个版本: **calculate-1.0** 和 **calculate-backup**。您希望查看它们是否完全相同。使用下述命令:

```
cmp calculate-1.0 calculate-backup
```

如果这两个文件相同, 那么您看不到任何东西。如果这两个文件不匹配, 那么您将看到一个类似于下面内容的消息:

```
calculate-1.0 calculate-backup differ: byte 31, line 4
```

正如图 17-1 所示，还有其他几个程序可以用来比较文件(**comm**、**diff**、**diff3** 和 **sdiff**)。所有这些程序都可以处理文本文件。也就是说，它们期望输入文本行：每行包含 0 个或者多个常规字符(字母、数字、标点符号、空白符)并以新行字符结尾。

因为 **cmp** 每次一个字节地比较文件，所以它不关心文件包含什么类型的数据。因此，可以使用 **cmp** 比较任何类型的文件：文本文件或者二进制文件。例如，上面的例子就是比较两个包含可执行程序的二进制文件。另外还可以比较两个音乐文件、两张图片、两个字处理程序文档等等(我们将在第 19 章和第 23 章中讨论文本文件和二进制文件)。

17.3 比较有序文本文件：comm

相关过滤器：**cmp**、**diff**、**sdiff**

comm 程序一行一行地比较两个有序的文本文件。当有两个相似的文件，并且希望查看它们之间的区别时，可以使用 **comm** 程序。**comm** 程序的语法为：

```
comm [-123] file1 file2
```

其中 *file1* 和 *file2* 是有序文本文件的名称。

comm 程序的漂亮之处在于它允许查看两个文件之间的区别。该程序以 3 列显示输出：第一列包含只在第一个文件中有的行；第二列包含只在第二个文件中有的行；第三列包含两个文件中都有的行。下面举一个例子说明。

两个亲密的朋友 Frick 和 Frack*在想他们拥有多少个共同的朋友。他们每人制作了一个他们各自朋友的列表，将列表键入到文件中，然后使用 **sort** 命令(参见第 19 章)将文件排序。最后他们使用 **comm** 比较这两个文件。

Frick 的朋友的有序列表位于文件 **frick** 中：

```
Alison Wonderland
Barbara Seville
Ben Dover
Chuck Wagon
Noah Peel
```

Frack 的朋友的有序列表位于文件 **frack** 中：

```
Alison Wonderland
Barbara Seville
Candy Barr
Chuck Wagon
Noah Peel
Sue Perficial
```

* Frick 和 Frack 是两位著名的喜剧溜冰者 Werner Groebli 和 Hans Mauch 的舞台名称。Groebli 和 Mauch 于 1937 从他们的祖国瑞士来到美国，数十年来，他们举办了大量的表演，并根据这些表演后来创作了 Ice Follies。他们最有名的技巧就是“cantilever spread-eagle，雄鹰展翅”(相关图片请在网络上搜索)。

Frick 和 Frack 使用下述命令比较这两个列表:

```
comm frick frack
```

输出为:

```

        Alison Wonderland
        Barbara Seville
Ben Dover
    Candy Barr
        Chuck Wagon
        Noah Peel
        Sue Perficial
```

注意输出有 3 列。第一列只有一个姓名(Ben Dover), 这说明第一个文件(**frick**)中只有一行是独有的。第二列有两个姓名(Candy Barr、Sue Perficial), 这说明第二个文件(**frack**)中有两行是独有的。第三列有 4 个姓名(Alison Wonderland、Barbara Seville、Chuck Wagon、Noah Peel), 这说明两个文件中有 4 行是重复的。因此, Frick 和 Frack 有 4 个朋友是相同的。

在这个例子中, 文件都是很小的——Frick 和 Frack 总共拥有 7 位朋友, 您可能奇怪他们为什么还要创建两个文件, 对文件排序, 运行 **comm** 命令, 然后再解释输出。难道 Frick 和 Frack 不能相互询问: 您认识 Alison 吗? 您认识 Barbara 吗? 等等。

答案是当然可以, 但是这只是一个人为设计的例子。如果使用的有序文件有数百行或者数千行——例如客户记录、统计数据或 MP3 播放器的一长串歌曲列表呢? 在这种情况下, 手工比较列表几乎不可能。必须拥有一个像 **comm** 这样的程序。

实际上, **comm** 在比较一个有序文件的两个有细小区别的版本(或许因为一些小错误而生成的区别), 并且还希望查看它们之间的区别时特别有用。例如, 假设您有两个非常长的有序文件, 这两个文件都是由数字构成的。在这两个文件的某些地方, 可能有些位置上的数字不一致。使用 **comm** 命令比较这两个文件, 在输出的中间, 将看到:

```

01023331
01023340
01023356
01023361
01023362
01023378
01023391
01023401
```

这指示了两个文件不匹配的地方的准确位置。

为了控制输出, **comm** 允许使用 **-1**、**-2** 和 **-3** 选项分别抑制掉第一列、第二列和第三列的输出。例如, 在我们的上一个例子中, 可以使用 **-3** 选项抑制掉第三列输出, 这将消除所有不必要的输出:

```
01023361
```


01023362

现在，您所看到的都是两个文件之间的不同，这正是您希望看到的全部内容。想象一下，如果文件包含有几千个数字，这会节省多少时间。

如果希望抑制掉不止一列的内容，则只需组合使用选项。例如，考虑上面讨论的 Frick 和 Frack 的朋友列表。假设 Frack 只希望显示那些属于他自己但是不属于 Frick 的朋友。他需要做的就是抑制掉第一列和第三列：

```
comm -13 frick frack
```

输出只显示那些在第二个文件中独有的行：

```
Candy Barr
Sue Perfficial
```

提示

comm 不按期望工作的最常见的原因就是输入文件没有排序。

如果需要比较两个不希望进行排序的文件——例如不希望将数据搞乱，则不能使用 **comm**。不过，可以使用 **diff**(本章后面讨论)。当比较某程序的不同版本的源代码时就是这种情况。

17.4 比较无序文本文件：diff

相关过滤器：**cmp**、**comm**、**sdiff**

comm 程序可以直观地示范两个文本文件的区别。但是，**comm** 程序有两个限制。首先，输入文件必须是有序的，而这在许多情况下是不可能的。例如，假设您有两个不同版本的长文件，例如计算机程序或者论文，而且您希望知道它们之间的区别。因为程序或者论文的各行并不是有序的，所以不能使用 **comm**。当然，您可以先对文件排序，但是这样的话，输出就没有意义了：您可以找出两个文件之间的区别，但是却失去了上下文关系。

此外，当比较那些小的，甚至是中等大小的文件时，**comm** 的输出非常好，但是当比较大的文件时，可能会产生混乱。这里又一次涉及到，文件的上下文关系的问题。当比较大的文件时，重要的是输出不仅能够显示出区别，而且还要显示出位置，这样就可以很容易地查找到不同的行。

设计 **diff** 程序的目的是克服这些限制。因此，当需要(1)比较无序文件，(2)比较大的文件时，可以使用 **diff**。更通用地讲，**diff** 可以用来查找任何类型的工作带来的区别，在这些工作中，时不时地会进行增量添加、删除或者改变。例如，多年以来，在程序修改后，程序员使用 **diff**(或者类似于 **diff** 的工具)跟踪程序各种版本之间的变化。

在开始之前，我必须警告您，除非您习惯了 **diff** 的输出，否则它的输出看上去有点神秘。但是，您一定会习惯它的。无论如何，**diff** 都是一个功能强大并且有用的程序，重要的是您要学会使用它，特别是当您是一名程序员时。

diff 的语法如下所示：

```
diff [-bBiqswy] [-c|-Clines|-u|-Ulines] file1 file2
```

其中, *file1* 和 *file2* 是文本文件的名称, *lines* 是说明上下文关系的行号。

当比较的两个文件相同时, **diff** 不显示任何输出(与 **cmp** 相似)。如果两个文件不同, 那么 **diff** 默认情况下将显示一组指示, 如果遵循这些指示, 将把第一个文件修改为第二个文件。下面举一个例子。

我们有两个文件。old-names 文件中包含:

```
Gene Pool  
Will Power  
Paig Turner  
Mark Mywords
```

new-names 文件中包含:

```
Gene Pool  
Will Power  
Paige Turner  
Mark Mywords
```

我们注意到这两个文件之间的唯一区别就是第三行上“Paige”的拼写。为了比较这两个文件, 我们使用:

```
diff old-names new-names
```

输出为:

```
3c3  
< Paig Turner  
---  
> Paige Turner
```

正如前面解释的, **diff** 的目的就是显示将第一个文件修改为第二个文件所需遵循的指示。这些指示的语法相当简洁, 但并不简单, 需要一定的练习才能理解它。但是, 我真的希望您能够熟悉这些类型的指示, 因为它们都是 Unix 文化的一个标准组成部分。实际上, 在许多场合都会遇到这种类型的语法, 不只是在使用 **diff** 的过程中。

diff 的输出使用 3 个不同的单字符指示: **c**(change, 改变)、**d**(delete, 删除)和 **a**(append, 追加)*。在上述例子中, 只看到了一个 **c** 指示。这意味着, 为了将第一个文件转换成第二个文件, 只需进行一个修改, 一个简单的改变。

c、**d**、**a** 每个字符的左边和右边都有一串行号。可能是一个单独的行号(例如上面的 3), 也可能是一串行号(例如 16、18)。左边的数字指第一个文件中的行, 右边的数字指第二个文件中的行。在上面的例子中, **3c3** 告诉我们, 将第一个文件中的第 3 行改变成第二个文件中的第 3 行。

* 为什么只有这 3 种指示呢? 对于两个相当相似的文件来说, 通过改变、删除和追加操作总是可以将一个文件转换成另一个文件。好好地想一想, 直至您明白其中的含义。

无论何时，当 **diff** 要求改变时，它就给出每个文件的实际行。第一个文件中的行由一个<(小于号)字符标记。第二个文件中的行由一个>(大于号)字符标记。出于可读性考虑，两组行被由若干个连字符(=)构成的直线分隔开。

下面考虑另一个例子，在这个例子中，**old-names** 文件包含：

```
Gene Pool
Paige Turner
Mark Mywords
```

new-names 文件包含：

```
Gene Pool
Will Power
Paige Turner
Mark Mywords
```

在这个例子中，两个文件之间唯一的区别就是第一个文件中不包含姓名“Will Power”。当使用 **diff** 比较这两个文件时，输出为：

```
1a2
> Will Power
```

这说明了如何将第一个文件改变为第二个文件。所需做的就是第一个文件中追加一行。具体而言，就是在第一个文件的第 1 行之后追加第二个文件中的第 2 行。注意，**diff** 命令给出了要追加的行的具体内容。>字符说明这一行来自第二个文件。

现在，考虑第三个例子，在这个例子中，**old-names** 文件包含：

```
Gene Pool
Will Power
Paige Turner
Mark Mywords
```

new-names 文件包含：

```
Gene Pool
Will Power
Paige Turner
```

两个文件之间的区别就是第二个文件不包含姓名“Mark Mywords”。当使用 **diff** 时，输出为：

```
4d3
< Mark Mywords
```

在这个例子中，**diff** 说明为了将第一个文件转换成第二个文件，需要删除第一个文件中的第 4 行。同样，实际行也被显示出来。<字符说明这一行位于第一个文件中(记住，**diff** 的目的就是说明如何将第一个文件转换成第二个文件)。

注意：在 **d** 命令中，通常可以忽略 **d** 之后的数字(在这个例子中为 3)。它说明 **diff** 在

第二个文件的什么地方发现了区别。

在结束本部分讨论之前，我先示范一下 **diff** 如何在更现实的例子中应用。考虑下述两个文件，每个文件都包含一些 Perl 脚本代码(现在不要关心这些代码能做什么，只需关注 **diff** 命令的输出)。第一个文件 **command-1.01.pl** 中包含：

```
# Check for illegal content
# in order to prevent spam
# If the address contains a URL, abort
if ($required("address") =~ m/(http):\// ) {
    $error_count += 1;
}
```

第二个文件 **command-1.02.pl** 中包含：

```
# If the address contains a URL, abort
if ($required("address") =~ m/(http|https|ftp):\// ) {
    $error_count += 1;
    &error_exit ("No URLs allowed");
}
```

下述 **diff** 命令比较这两个文件：

```
diff command-1.01.pl command-1.02.pl
```

输出为：

```
1,2d0
< # Check for illegal content
< # in order to prevent spam
4c2
< if ($required("address") =~ m/(http):\// ) {
---
> if ($required("address") =~ m/(http|https|ftp):\// ) {
5a4
> &error_exit ("No URLs allowed");
```

输出的解释有两种方式。从字面上讲，它说明将第一个文件转换成第二个文件应该遵循什么指示：

- 删除第一个文件中的第 1 行和第 2 行。
- 将第一个文件中的第 4 行修改成第二个文件中的第 2 行。
- 在第一个文件中第 5 行之后追加第二个文件中的第 4 行。

解释该输出的一个更好的方法就是能够读懂它，并且能够立即清晰地理解两个文件之间的区别。当然，这就是为什么首先学习使用 **diff** 的原因。关键在于必须能够读懂 **c**(change, 改变)、**d**(delete, 删除)和 **a**(append, 追加)命令，并且能够立即掌握它们的含义。可以想象出，这需要多加练习。但是最终，您将能够快速不费力地读懂并理解这样的输出。

17.5 diff 使用的选项

diff 程序相当复杂，它有大量的选项和许多种生成输出的方式。在本节及下一节中，将讨论一些最重要的选项。完整的说明，请参见系统的说明书页(**man diff**)。

首先介绍少数几个选项，这些选项告诉 **diff** 在进行比较时忽略特定的区别。**-i**(case insensitive, 不区分大小写)选项告诉 **diff** 忽略大写字母和小写字母之间的任何区别。例如，当使用**-i**选项时，**diff** 认为下面3行都是相同的：

```
This is a BIG test.
this is a big test.
THIS IS A BIG TEST.
```

-w 和 **-b** 选项允许控制 **diff** 使用空白符(空格和制表符)的方式。当希望忽略格式化数据中的空格和制表符时，使用这些选项相当方便。**-w**(whitespace, 空白符)选项忽略所有的空白符。例如，使用了**-w**选项之后，下述两行是相同的。

```
XX
X      X
```

-b 选项与此相似，但是它不忽略所有的空白符，而只忽略空白符数量上的区别。例如，如果使用**-b**选项，则上述两行是不相同的，因为第二行中有空白符，而第一行中没有。但是，下述两行是相同的：

```
X  X
X      X
```

这是因为这两行都有空白符，它们之间的区别只是空白符的数量。**-w** 和 **-b** 选项之间的区别比较微妙，因此如果您遇到了空白符问题，并且感到迷惑，可以试试这两种选项，看看哪一种更合适。

-B(blank lines, 空白行)选项告诉 **diff** 忽略所有的空白行。例如，假设您有两个文件，分别包含了您所写的论文的不同版本。您希望比较它们，但是一个副本各行间有一个空白行，另一个副本各行间有两个空白行。如果使用**-B**选项，那么 **diff** 将忽略空白行，只查看文本行。

其余的 **diff** 选项控制 **diff** 如何显示结果。**-q**(quiet, 静止)选项告诉 **diff**，当两个文件不同时，省略所有的细节。例如，如果比较两个不同的文件 **frick** 和 **frack**，并且使用**-q**选项，那么看到的就是：

```
Files frick and frack differ
```

这样，当使用 **diff -q** 比较两个文件时，实质上和使用 **cmp**(本章前面讨论过)相同。两个命令之间最大的区别就是 **diff** 只比较文本文件，而 **cmp** 可以比较任何类型的文件。

正如前面所述，当 **diff** 发现两个文件相同时，它不显示任何东西。大多数 Unix 程序都是这种情况：当它们没有什么事可说时，它们就不说什么。但是，有时候也希望明确提

示两个文件相同。在这种情况下，可以使用**-s**(same, 相同)选项。例如，如果您比较两个文件 **frick** 和 **frack**，而且这两个文件相同，那么通常您看不到任何结果。但是，如果使用了 **-s** 选项，您将看到：

```
Files frick and frack are identical
```

17.6 比较文件时的输出格式：diff、sdiff

正如前面讨论的，当 **diff** 比较两个文件时，默认输出由指示(**c**、**d**、**a**)和行号构成。如果遵循这些指示，可以将第一个文件转换成第二个文件。这种类型的输出拥有一个优点，即简洁，一旦习惯了这种输出，它实际上就是可读的。从我的经验来看，大多数时候您需要的就是这些。

但是，默认格式的缺点在于当您习惯了这种格式后，就会在您的大脑的灰质区形成不可逆转的变化。当阅读输出时，最大的问题就是极少有上下文关系。您看到的就是一些行号，以及要改变的行。基于这一原因，**diff** 提供了 3 种选项(**-c**、**-u** 和 **-y**)，以生成更易阅读的输出类型。另外，还有一个程序 **sdiff**，可以并排地比较两个文件。下面介绍细节。

对 **diff** 使用 **-c**(context, 上下文关系)选项将以不太简洁，但是更易理解的格式(相对于默认输出)显示两个文件之间的区别。使用了 **-c** 选项之后，**diff** 将不再显示指示和行号，而是显示存在不同的实际行，同时还显示不同行的上面和下面各两行内容。下面举例说明。

假设您有两个要比较的文件。第一个文件是 **smart-friends**，其中包含：

```
Alba Tross
Dee Compose
Pat D. Bunnie
Phil Harmonic
```

第二个文件是 **rich-friends**，其中包含：

```
Alba Tross
Dee Compose
Mick Stup
Pat D. Bunnie
```

首先，我们按常规方式比较这两个文件：

```
diff smart-friends rich-friends
```

该输出简洁，但是有点神秘：

```
2a3
> Mick Stup
4d4
< Phil Harmonic
```


现在，我们使用-c 选项：

```
diff -c smart-friends rich-friends
```

该输出比较长，但是易于理解：

```
*** smart-friends 2009-02-14 15:33:50.000000000 -0700
--- rich-friends 2009-02-14 15:34:04.000000000 -0700
*****
*** 1,4 ****
    Alba Tross
    Dee Compose
    Pat D. Bunnie
-   Phil Harmonic
--- 1,4 ----
    Alba Tross
    Dee Compose
+   Mick Stup
    Pat D. Bunnie
```

最顶端的两行展示两个文件的信息。第一个文件由*(星号)字符标记，第二个文件由-(连字符)标记。在这两行之后，是每个文件的摘录，它们准确地示范了为了使两个文件相同，需要进行些什么改变。

尽管这种格式相对于默认输出而言更易于理解，但是它拥有一个明显的缺点：因为 diff 显示两个文件的摘录，所以输出中有重复的文本，需要生成大量的输出。考虑我们的例子，该例只比较两个短文件，并且只有简单的区别。您可以想象，如果比较两个有许多区别的大文件，所生成的输出会有多长。在这种情况下，-c 选项所生成的输出要比默认格式生成的输出冗长得多。

作为妥协，可以使用-u(unified output, 统一输出)选项。这时生成的输出类似于-c 选项生成的输出，但是没有重复行。例如，当使用下述命令时：

```
diff -u smart-friends rich-friends
```

输出为：

```
--- smart-friends      2009-02-14 15:33:50.000000000 -0700
+++ rich-friends      2009-02-14 15:34:04.000000000 -0700
@@ -1,4 +1,4 @@
    Alba Tross
    Dee Compose
+Mick Stup
    Pat D. Bunnie
-Phil Harmonic
```

最后一种输出选项生成并排的格式，在这种格式中，第一个文件中的每一行都与第二个文件中的对应行并排着显示。为了使用该格式，可以使用-y 选项：

```
diff -y smart-friends rich-friends
```

提示

默认情况下, 当对 **diff** 使用 **-c** 或者 **-u** 选项时, 输出在每个不同点的上方和下方分别显示两行上下文。

如果希望显示不同数量的上下文行, 可以使用 **-C**(大写字母“C”)替代 **-c**, **-U**(大写字母“U”)替代 **-u**。使用 **-C** 或 **-U**, 后面跟着希望的额外行的数量即可, 例如:

```
diff -C5 file1 file2
diff -U3 file1 file2
```

并排的输出去看上去如下所示:

```
Alba Tross      Alba Tross
Dee Compose     Dee Compose
                > Mick Stup
Pat D. Bunnie   Pat D. Bunnie
Phil Harmonic <
```

您可以看出这种类型的输出的优点是: 非常容易看出区别。例如, 在我们的例子中, 非常明显, 有 3 个姓名在两个文件中都存在(Alba、Dee 和 Pat), 1 个姓名只位于第二个文件中(Mick), 1 个姓名只位于第一个文件中(Phil)。当然, 这种格式的缺点就是对于长文件来说, 可能生成许多输出。

如果喜欢这种类型的输出, 则还有一个特殊用途的程序 **sdiff**(side-by-side diff, 并排 diff), 您可以使用它替代 **diff -y**。例如, 下述两条命令生成相同的输出:

```
diff -y smart-friends rich-friends
sdiff smart-friends rich-friends
```

当需要进行并排比较时, 许多人愿意使用 **sdiff**, 因为它有许多专用的选项, 可以提供大量的控制。**sdiff** 的语法为:

```
sdiff [-bBilsW] [-w columns] file1 file2
```

其中 *file1* 和 *file2* 是文本文件的名称, *columns* 是列宽。

sdiff 的使用相当简单。例如, 为了比较上面例子中的两个文件, 我们可以使用:

```
sdiff smart-friends rich-friends
```

输出为:

```
Alba Tross      Alba Tross
Dee Compose     Dee Compose
                > Mick Stup
Pat D. Bunnie   Pat D. Bunnie
Phil Harmonic <
```

正如前面所述, **sdiff** 拥有许多选项。我们将介绍最重要的选项, 其中一些选项与 **diff**

的选项相同。为了阅读其余的选项，请查看系统的说明书页(`man sdiff`)。

首先，有几种选项允许减少不必要的输出的数量。首先是-l(小写的“L”)选项，当两个文件拥有共同行时，该选项只显示左边的列。例如，如果使用：

```
sdiff -l smart-friends rich-friends
```

那么输出为：

```
Alba Tross      (  
Dee Compose    (  
                > Mick Stup  
Pat D. Bunnie  (  
Phil Harmonic  <
```

-s(same, 相同)选项更进一步减少输出：它告诉 `sdiff` 不显示在两个文件中相同的任何行。例如：

```
sdiff -s smart-friends rich-friends
```

该输出是最少的，并且易于理解：

```
                > Mick Stup  
Phil Harmonic <
```

当使用的文件所拥有的行都是短行时(就如上面的例子)，通常会发现 `sdiff` 所使用的默认列宽太宽。当发生这种情况时，可以使用-w 选项改变列的宽度。只需在-w 选项之后输入希望每列所拥有的字符数即可。例如：

```
sdiff -w 30 smart-friends rich-friends
```

当然，也可以将各个选项组合起来使用。我最喜欢的方式就是使用-s 和-w 30 选项。例如：

```
sdiff -s -w 30 smart-friends rich-friends
```

看到输出后，还可以调整列的宽度以适应数据。

最后，`sdiff` 还有 4 个选项类似于 `diff` 中使用的选项。-i 选项忽略大写字母和小写字母之间的区别；-W 选项忽略所有的空白符；-b 选项忽略空白符数量上的区别；-B 选项忽略空白行(注意 `sdiff` 使用-W，而 `diff` 使用-w。这一区别有其历史原因，且一直没有改变过来)。

17.7 差分和补丁

多年以来，`diff` 一直是一个非常重要的工具，程序员使用它来掌握不同版本的程序之间的区别。例如，假设您是一名程序员，您在开发一个 C 程序 `Foo`。当前的版本是 2.0，它存储在文件 `foo-2.0.c` 中。现在，您正在开发 2.1 版，它存储在 `foo-2.1.c` 中。一旦 2.1 版本完成，就可以使用下述命令查看两个版本之间的变化：

```
diff foo-2.0.c foo-2.1.c > foo-diff-2.1
```

现在输出文件(**foo-diff-2.1**)包含一系列指示, 如果遵循这些指示, 就可以将 **foo-2.0.c** 转换成 **foo-2.1.c**。

通常, 将一个文件转换成另一个文件的一串指示称为一个差分(diff)。因此, 我们可以说 **foo-diff-2.1** 中包含将 **foo-2.0.c** 转换成 **foo-2.1.c** 的差分。

程序员创建差分有两个原因: 在备份时节省存储空间, 以及将变化发布给他人。

假设 Foo 是一个非常大的程序。所以该程序的每个版本都要谨慎地做个备份, 但是这样要占用大量的存储空间。另一种方法就是对基准版本(**foo-2.0.c**)进行完整的备份。然后只备份差分, 例如 **foo-diff-2.1**、**foo-diff-2.2** 等, 这些差分占用的存储空间就小多了(当到达 3.0 版时, 可以再保存一个完整的副本)。

假设您现在正在开发 2.7 版, 但是由于一个突发灾难事件使原始文件丢失。为了从备份中恢复程序, 首先复制基准文件 **foo-2.0.c**, 然后使用第一个差分重新创建 **foo-2.1.c**, 使用第二个差分重新创建 **foo-2.2.c**, 依次类推, 一直到重新创建 **foo-2.7.c**。换句话说, 通过恢复基准副本和一系列差分, 可以重新创建所有版本的程序。实际上, 通过使用这种技术, 可以恢复存储在文本文件中的不同版本的任何东西, 包括小说、论文、销售演示文档等。

当以这种方式使用差分——以一个文件重新创建另一个文件时, 我们称应用了差分。用来应用差分的程序称为 **patch**(其细节已经超出了本书的范围)。在我们的例子中, 为了重新创建丢失的文件, 需要从备份中复制基准版本和所有的差分, 然后使用 **patch** 一个接一个地应用差分。

程序员使用差分的第二种方式就是分发程序的变化。例如, 假如说有许多人拥有 Foo 程序版本 2.0 的源代码。每个人下载并安装该程序需要花费一会时间, 但是最终他们现在有这个程序了。当您准备发布版本 2.1 时会怎么做?

您可以要求每个人下载完整的新程序。但是, 这需要花费很长的时间。实际上, 您只需发布差分, 这就比较小了。为了改变到 2.1 版, 所有的用户只需使用 **patch** 应用差分即可。如果基于某些原因, 他们在安装新版本程序时出现了问题, 那么他们还可以使用 **patch** 反应用该差分, 使程序又回到 2.0 版。

当程序员以这种方式使用差分时, 通常将差分称为补丁(patch)。因此, 在我们的例子中, 可以说您发布了一个 2.1 版的补丁, 用户使用 **patch** 程序应用该补丁。

以差分方式发布变化的优点在于, 应用补丁要比下载并安装全新版本的程序快许多。实际上, 在 Internet 初期, 下载速度非常慢, 升级大型程序的唯一实用方式就是发布补丁, 然后用户再自己应用补丁。

在 Unix 初期, 程序员通常使用 **diff** 和 **patch** 来维护、备份及发布程序。但是, 很长一段时间以来, 由于有更好的系统来自动执行这一任务, 很少有程序员直接使用 **diff** 和 **patch**。实际上, 他们使用一个复杂的版本控制系统(version control system), 该系统有时候称为源代码控制系统(source code control system, SCCS)或者修订控制系统(revision control system, RCS)。这样的系统通常由软件开发人员和工程师使用, 管理大型程序、文档、蓝图等的开发。实际上, 如果没有现代的版本控制系统, 那么大型团队的人员不可能在创造性的项目中一起工作。

但是，无论复杂程度如何，所有的版本控制系统都依赖于创建、发布和应用差分这些基本概念。这就是这些基本概念之所以重要的原因。

名称含义

差分

单词 **diff**(差分)来源于 **diff** 程序，该程序用于比较两个文件。在 Unix 人士中，经常将“diff”作为名词或动词使用。

例如，您可能听到有人说“Send me your diffs for the Foo program(将 Foo 程序的差分发送给我)”，这句话的含义就是“将包含 Foo 程序更新的文件发送给我。”

有时候也可能听到有人将 **diff** 作为动词使用：“If you want to see the changes I made to your news article, just diff the two files(如果您想知道我对您的新闻文章所做的修改，只需使用 **diff** 比较两个文件即可)。”

17.8 抽取数据列：cut

相关过滤器：**colrm**、**join**、**paste**

cut 程序是一个从数据中抽取指定列并将其他内容抛弃的过滤器(这与 **colrm** 相反，**colrm** 从数据中删除指定列并保存其他内容)。

cut 程序拥有极大的灵活性。它既可以从每行中抽取特定列，也可以从每行中抽取指定的区域(称为字段)。如果您是一名数据库专家，那么您可以考虑使用 **cut** 完成关系映射(如果您不是数据库专家，那么不用担心，您的生活仍是完整的)。

在本节中，将集中讨论如何使用 **cut** 抽取数据列。在下一节中，将讨论如何从数据中抽取字段。

cut 的语法(当抽取数据列时)为：

```
cut -c list [file...]
```

其中 *list* 是要抽取的列的列表，*file* 是输入文件的名称。

使用列表告诉 **cut** 希望抽取哪些列时，可以指定一个或多个列号，各个列号用逗号隔开。列表中不能含有任何空格。例如，如果希望抽取第 10 列，则使用 **10**。如果希望抽取第 1、8 和 10 列，则使用 **1,8,10**。

另外还可以指定列的范围，即将开头与结尾的列号用连字符连起来。例如，为了抽取第 10 列至第 15 列，可以使用 **10-15**。为了抽取第 1、8 及 10 至 15 列，可以使用 **1,8,10-15**。

下面举一个如何使用 **cut** 的例子。假设您有一个文件 **info**，该文件中包含一组人员的相关信息。每行包含一个人的数据。其中，列 14-30 包含的是姓名，列 42-49 包含的是电话号码。下面是一些样本数据：

```
012-34-5678 Ambercrombie, Al 01/01/72 555-1111
123-45-6789 Barton, Barbara 02/02/73 555-2222
234-56-7890 Canby, Charles 03/03/74 555-3333
```

```
345-67-8901 Danfield, Deann 04/04/75 555-4444
```

为了只显示姓名，可以使用：

```
cut -c 14-30 info
```

您将看到：

```
Ambercrombie, Al
Barton, Barbara
Canby, Charles
Danfield, Deann
```

为了显示姓名和电话号码，可以使用：

```
cut -c 14-30,42-49 info
```

您将看到：

```
Ambercrombie, Al 555-1111
Barton, Barbara 555-2222
Canby, Charles 555-3333
Danfield, Deann 555-4444
```

如果您喜欢，还可以省略 **-c** 之后的空格。实际上，大多数人都是这样做的。因此，下述命令与前面最后一条命令等价：

```
cut -c14-30,42-49 info
```

返回我们的例子，通过重定向标准输出到文件，可以保存该信息。例如：

```
cut -c 14-30,42-49 info > phonelist
```

cut 程序可以方便地应用于管道线中。下面举一个例子。您共享一台多用户 Linux 计算机，并且希望生成一个当前登录到系统的用户标识列表。因为一些用户标识可能登录不止一次，所以您还希望显示每个用户标识登录了多少次。

首先使用 **who** 命令(参见第 8 章)。该命令将生成一个报告，登录到系统中的每个用户标识占据一行。下面是一个典型结果：

```
harley console Jul 8 10:30
casey ttypl Jul 12 17:46
weedly tty4 Jul 12 21:22
harley tty0 Jul 12 16:45
linda tty3 Jul 12 17:41
```

可以看出，用户标识在第 1 列至第 8 列中显示。因此，我们可以使用下述命令抽取用户标识：

```
who | cut -c 1-8
```

输出为：


```
harley
casey
weedly
harley
linda
```

下面，我们更进一步，使用 `sort` 命令对用户标识列表排序，并使用 `uniq -c` 对重复行进行统计(`sort` 和 `uniq` 命令都在第 19 章中解释)。将这些东西连在一起，得到：

```
who | cut -c 1-8 | sort | uniq -c
```

注意，在管道线中使用选项不存在任何问题。输出为：

```
1 casey
2 harley
1 linda
1 weedly
```

作为该管道线的一个有趣的变体，我们问一个问题：如何显示所有正好登录系统两次的用户标识的名称？解决方法就是搜索 `uniq` 的输出中所有包含“2”的行。这一步可以使用 `grep`(参见第 19 章)完成：

```
who | cut -c 1-8 | sort | uniq -c | grep "2"
```

输出为：

```
2 harley
```

提示

为了重新安排表中的各列，可以在 `cut` 命令之后使用 `paste`。

17.9 记录、字段和定界符；抽取数据字段：cut

在上一节中，我们示范了如何使用 `cut` 程序抽取指定的数据列。但是，`cut` 还有另外一种应用：它可以抽取数据字段。为了理解这种工作方式，需要首先理解几个基本思想。

考虑下面两个不同的文件。第一个文件包含下述各行：

```
Ambercrombie Al 123
Barton Barbara 234
Canby Charles 345
Danfield Deann 456
```

第二个文件包含：

* 严格地讲，这个 `grep` 命令将查找任何包含字符“2”的行。例如，如果某人登录了 12 次或 20 次，那么也会查找出这个人。一种更好的解决方法，即我们将在第 20 章中讨论的技巧，就是使用命令 `grep "\<2>"`。该命令仅限于查找那些只包含“2”的行。

```
Ambercrombie:Al:123
```

```
Barton:Barbara:234
```

```
Canby:Charles:345
```

```
Danfield:Deann:456
```

在两个文件中，每行都包含一个名字、一个姓氏和一个标识号，实际上，这两个文件包含相同的信息。但是，它们之间存在很大的区别。

第一个文件容易阅读，因为信息按列对齐。第二个文件更适合于程序读取，因为:(冒号)字符将每行分成 3 部分。使用第 12 章中讨论的术语，我们可以称第一个文件是人类可读的，而第二个文件是机器可读的。

当使用由程序处理的数据时，我们经常会遇到机器可读的文件，与上面的第二个例子相似。对于这种数据来说，每行都称为一个记录(record)，每行的各个部分称为字段(field)，而充当字段分隔符的字符称为定界符(delimiter)。在我们的例子中，总共有 4 条记录，每条记录有 3 个字段(姓、名、标识号)。在每个记录中，定界符都是冒号。

当然，定界符并不一定总是冒号。原则上，任何在实际数据中不出现的字符都可以用作定界符。最常见的定界符就是逗号、空格、制表符和空白符(也就是制表符和空格的组合)。

实际上，逗号经常用作定界符，而且还有一个特殊的名称来描述使用逗号作为定界符的数据。我们称这样的数据以 CSV(comma-separated value, 逗号分隔值)格式存储*。

或许使用定界符的最有趣的机器可读文件就是 Unix 口令文件(/etc/passwd)，Unix 口令文件在第 11 章中讨论过。系统中每个用户标识在口令文件中包含一行。在每一行中，各个字段使用:字符分隔。如果一个字段是空的，则会看到两个连续的:字符。

为了查看系统上的口令文件，需要使用下述命令之一*：

```
cat /etc/passwd
less /etc/passwd
```

现在我们已经打好了基础，下面示范如何使用 **cut** 程序从文件的行中抽取字段。**cut** 程序的语法为：

```
cut -c list [file...]
cut -f list [-d delimiter] [-s] [file...]
```

* 直到最近几年，CSV 格式还是程序之间交换数据的最流行的存储格式，特别是对电子表格程序而言，例如 Microsoft Excel。现在，XML(Extensible Markup Language, 扩展标记语言)是最广泛使用的存储格式，因为它能够处理许多类型的数据。CSV 格式，尽管易于理解，但是由于其只能用于纯文本，所以存在诸多限制。

出于参考目的(以防需要)，下面给出 CSV 格式的综合性技术定义：

“CSV 格式用来存储组织成记录的文本数据，每个记录都以一个新行字符(或者是 Windows 中的换行符)结束。在每个记录中，各个字段由逗号分隔。字段前面或者后面的任何空白符(空格或制表符)都被忽略。字段可以用双引号括起来，但是双引号通常都被忽略。如果字段中包含有逗号、双引号或者新行字符，又或者字段以空格或制表符开头或结尾，那么这个字段必须用双引号括起来。在字段中，双引号字符用两个连续的双引号表示。”

* 在很久以前的 Unix 版本中，口令(当然是加密的)存放在口令文件中，口令文件的名称也是因此而来的。在现代的 Unix 中，实际口令并没有存放在这个文件，正如第 11 章讨论的，出于安全考虑，加密的口令存储在一个不同的文件中，这个文件就是/etc/shadow，我们称这个文件为影子文件。

其中, *list* 是抽取字段的列表, *delimiter* 是分隔字段所使用的定界符, 而 *file* 是输入文件的名称。

与使用 **-c** 选项相比, 字段列表的格式完全相似。您可以指定一个或多个数字, 各个数字用逗号隔开。列表中不能含有任何空格。例如, 为了只抽取字段 10, 可以使用 **10**。为了抽取字段 1、8 和 10, 可以使用 **1,8,10**。

另外还可以指定字段的范围, 即将开头和末尾的字段用连字符连起来。例如, 为了抽取字段 10 至 15, 可以使用 **10-15**。为了抽取字段 1、8 以及 10-15, 可以使用 **1,8,10-15**。

下面举一个例子。在口令文件(*/etc/passwd*)中, 每行的第一个字段就是用户标识。假定您希望查看系统中所有已注册的用户标识。记住这个文件使用:作为定界符, 您所需做的全部就是从口令文件中的每一行抽取第一个字段。命令为:

```
cut -f 1 -d ':' /etc/passwd
```

如果希望排序列表, 只需将输出管道传送给 **sort**(参见第 19 章)即可:

```
cut -f 1 -d ':' /etc/passwd | sort
```

下述例子从同一个文件中抽取字段 1、3、4 和 5:

```
cut -f 1,3-5 -d ':' /etc/passwd | sort
```

请注意, 在这个命令中, 我将定界符(:)引用起来。这是一种好习惯, 可以确保在 **shell** 解析命令时不会错误地解释定界符。在这个例子中, 省略引号也没有问题, 但是如果定界符是空格、制表符或者元字符, 则必须引用定界符。

如果 **cut** 遇到不包含任何定界符的行, 会发生什么情况呢? 默认情况下, 这样的行只是简单地写入标准输出。如果希望抛弃这样的行, 可以使用 **-s(suppress, 抑制)** 选项。

最后一点。与上一节中讨论的 **-c** 选项一样, 您可以省略 **-f** 和 **-d** 之后的空格。因此, 下面的命令与前面两个例子等价:

```
cut -f1 -d':' /etc/passwd | sort
cut -f1,3-5 -d':' /etc/passwd | sort
```

大多数有经验的 Unix 人士都省略空格。

提示

当希望从既有定界符, 而且列宽又固定的文件中抽取字段时, 可以使用 **cut -d** 或者 **cut -c**。在这些情况中, 您将会发现使用定界符(**-d**)是一个较好的选择, 因为它不易出错。

17.10 组合数据列: paste

相关过滤器: **colrm**、**cut**、**join**

paste 程序组合数据列。这个程序拥有极大的灵活性。**paste** 程序可以将几个文件(其中每个文件都包含一系列数据)组成一个大表。另外, 也可以将连续的数据行组合起来, 构建多

个列。在本节中，我们将集中讨论 **paste** 程序最有用的特性：组合分离的文件。如果希望了解 **paste** 更多的细节问题，可以查看该程序的说明书页(**man paste**)。

paste 程序的语法为：

```
paste [-d char...] [file...]
```

其中 *char* 是用作分隔符的字符，而 *file* 是输入文件的名称。

使用 **paste** 可以将各数据列组合成一个大表。如果需要，还可以通过重定向标准输出将这个表保存到文件中。下面举一个例子。假设您有 4 个文件 **idnumber**、**name**、**birthday** 和 **phone**。各个文件的内容如下所示。

文件 **idnumber** 包含：

```
012-34-5678
123-45-6789
234-56-7890
345-67-8901
```

文件 **name** 包含：

```
Ambercromby, Al
Barton, Barbara
Canby, Charles
Danfield, Deann
```

文件 **birthday** 包含：

```
01/01/85
02/02/86
03/03/87
04/04/88
```

最后是文件 **phone**，其中包含：

```
555-1111
555-2222
555-3333
555-4444
```

您希望构建一个大文件 **info**，将所有这些数据组合成一个单独的表。在这个表中，每个文件的数据都应该位于自己的列中。所使用的命令为：

```
paste idnumber name birthday phone > info
```

info 文件的内容为：

```
012-34-5678  Ambercromby, Al  01/01/85  555-1111
123-45-6789  Barton, Barbara  02/02/86  555-2222
234-56-7890  Canby, Charles  03/03/87  555-3333
345-67-8901  Danfield, Deann 04/04/88  555-4444
```

注意这个输出中的空格数量有点奇怪。这是因为，默认情况下，**paste** 在每两列实体之间放一个制表符字符，而 Unix 假设制表符为每 8 个位置一个，且以位置 1 为起点。换句话说，就是 Unix 假定制表符被设置为位置 1、9、17、25 等(我们将在第 18 章中讨论 Unix 使用制表符的细节问题)。

为了告诉 **paste** 在各列之间使用一个不同的(非制表符)字符，可以使用 **-d(delimiter, 定界符)** 选项，后面跟一个括在单引号中的备选字符。例如，为了创建相同的表，但各列之间用空格隔开，可以使用：

```
paste -d ' ' idnumber name birthday phone
```

现在输出看上去如下所示：

```
012-34-5678 Ambercromby, Al 01/01/72 555-1111
123-45-6789 Barton, Barbara 02/02/73 555-2222
234-56-7890 Canby, Charles 03/03/74 555-3333
345-67-8901 Danfield, Deann 04/04/75 555-4444
```

如果指定了不止一个定界符，那么 **paste** 将轮流使用每个定界符，如果需要的话再重复使用各个定界符。例如，下述命令指定了两个不同的定界符，一个| (竖线) 和一个%(百分符号)。

```
paste -d ' |%' idnumber name birthday phone
```

输出为：

```
012-34-5678|Ambercromby, Al%01/01/85|555-1111
123-45-6789|Barton, Barbara%02/02/86|555-2222
234-56-7890|Canby, Charles%03/03/87|555-3333
345-67-8901|Danfield, Deann%04/04/88|555-4444
```

提示

可以认为 **paste** 与 **cat** 相似。两者之间的区别在于 **paste** 水平组合数据，而 **cat** 垂直组合数据。

通过使用 **cut** 和 **paste**，可以改变表中各列的顺序。例如，假设您有一个文件 **pizza**，该文件中包含您准备为聚会制作的 4 种不同比萨饼的信息：

```
mushrooms regular sausage
olives      thin    pepperoni
onions      thick   meatball
tomato      pan     liver
```

您希望改变第一列和第二列的顺序。首先，将每列保存到一个单独文件中：

```
cut -c 1-9 pizza > vegetables
cut -c 11-17 pizza > crust
cut -c 19-27 pizza > meat
```

现在按照自己希望的顺序，将这 3 列组合成一个单独的表：

```
paste -d ' ' crust vegetables meat > pizza
```

因为这是一个短文件，所以可以使用 **cat**(参见第 16 章中的讨论)显示它的内容：

```
cat pizza
```

现在数据看上去类似于：

```
regular mushrooms sausage
thin      olives      pepperoni
thick     onions      meatball
pan       tomato      liver
```

(当然，这是一个小的人为设计的例子，但是想一想，如果您需要交换一个拥有数百行或者数千行的文件中的各列，那么这一技能是多么的重要。)

一旦完成了自己希望的改变，则剩下两件事情需要完成。首先，使用 **rm** 程序(参见第 25 章)删除 3 个临时文件：

```
rm crust vegetables meat
```

其次，看看前来参加聚会的人是不是有人喜欢吃 **liver** 味和 **tomato** 味的比萨饼。

17.11 练习

1. 复习题

1. Unix 中有 10 个用于比较文件、排序文件以及从文件中选取数据的重要程序。这 10 个程序是什么？为什么有这么多的程序？
2. 默认情况下，**comm** 程序比较两个文件，并生成 3 列输出。解释每列的含义。如何抑制一个特定的列？
3. **diff** 和 **sdiff** 程序都用于比较文件。那么，什么时候使用 **diff** 呢？什么时候又使用 **sdiff** 呢？
4. 给您一个大的文本文件。使用哪些程序进行如下数据选取操作：a)重复行；b)唯一行；c)包含特定模式的行；d)以特定模式开头的行；e)数据列？

2. 应用题

1. 您热衷于和您的两个朋友 Claude 及 Eustace 比较喜爱的食物。创建 3 个文件：**me**、**claudio**、**eustace**。每个文件都包含 5 行有序的数据，每行都包含有食物的名称。仅使用两条 Unix 命令，显示在所有 3 个文件中都出现的食物列表(提示：可能需要创建一个临时文件)。

2. **comm** 程序用来比较两个有序文件，**diff** 程序用来比较无序文件。列举 3 种可以使用 **comm** 比较的数据类型。列举 3 种可以使用 **diff** 比较的数据类型。有没有一些情况，两个程序都可以使用呢？

3. Unix 的口令文件(/etc/passwd)中每行都包含一个用户标识的信息。在每行中，数据的不同字段用:(冒号)字符分隔。其中一个字段包含用户标识所使用 shell 的名称。使用下述命令显示并研究系统上口令文件的格式：

```
less /etc/passwd
```

(在 **less** 程序中，可以按<Space>键向下显示各页，或者按 **q** 键退出。)

什么命令可以用来读取口令文件并显示系统中所使用的各种 shell 的列表？如何对输出进行排序，从而使输出更容易阅读？如何消除重复行？

4. CSV 格式(逗号分隔值格式)描述一个包含机器可读数据的文件，在这种格式中数据的各个字段用逗号分隔。您有 5 个文件 **data1**、**data2**、**data3**、**data4** 和 **data5**，每个文件都包含一行数据。使用什么命令可以将这 5 列组合成一个 CSV 格式的文件 **csvdata**？如果某个文件中的行数少于其他文件中的行数，会发生什么情况呢？

3. 思考题

1. **diff** 程序的目的是通过显示将第一个文件转换成第二个文件的简洁指示，从而展示两个文件之间的区别。创建两个文件 **a** 和 **b**。比较下述命令的输出：

```
diff a b
diff b a
```

您可以发现什么模式？

2. 为什么 **diff** 的输出如此紧凑？它应该更易于理解吗？

3. 给您一个有 10000 行的文本文件。该文件包含两列数据，您必须改变两列的顺序。使用 **cut** 和 **paste** 命令可以快速准确地实现这个任务。假定没有这两条命令，您又该怎么做呢？假定您可以使用其他任意工具，那么如何实现这个任务呢？考虑一下使用文本编辑器、字处理程序、电子表格程序或者编写自己的程序等对此进行处理会是什么情况。您认为有比使用 Unix 的 **cut** 和 **paste** 程序组合更简单的方法吗？为什么呢？本章中所讨论的那些程序，又是什么使它们如此有用呢？

过滤器：统计和格式化

第16~19章这4章是讨论过滤器的，本章是其中的第3章。过滤器就是能够从标准输入读取文本数据并向标准输出写入文本数据(每次一行)的程序。在本章中，我们将讨论如何操作文本。我们将特别讨论如何使用行号，如何统计行、单词和字符的数量，以及如何用不同的方式格式化文本。

在讨论过程中，我们还将涉及几个非常有趣的话题，包括 Unix 如何处理制表符和空格，以及文本文件为什么经常是每行 80 个字符：这是一个非常有趣的故事，您可能都想象不到。

18.1 创建行号：nl

相关过滤器：**wc**

nl 过滤器提供一个简单但是有用的服务：它在文本中插入行号。**nl** 程序的语法为：

```
nl [-v start] [-i increment] [-b a] [-n ln|rn|rz] [file...]
```

其中 *start* 是起始号，*increment* 是增量，而 *file* 是文件的名称。

nl 程序在两种情形中派得上用场。第一种情形就是当希望在一些数据中永久插入行号，然后再保存时。第二种情形就是当希望在命令的输出中临时插入行号，以便于理解时。下面从介绍一个简单但是有用的例子入手。

假设您初次参与约会，您希望给对方留下一个良好的印象。为了准备约会，您创建了一个文件 **books**，其中包含您喜欢的书的列表：

```
Crime and Punishment
The Complete Works of Shakespeare
Pride and Prejudice
Harley Hahn's Internet Yellow Pages
Harley Hahn's Internet Insecurity
Harley Hahn's Internet Advisor
```

为了对这个列表进行编号，可以使用命令：

```
nl books
```

输出为：

```
1 Crime and Punishment
2 The Complete Works of Shakespeare
3 Pride and Prejudice
4 Harley Hahn's Internet Yellow Pages
5 Harley Hahn's Internet Insecurity
6 Harley Hahn's Internet Advisor
```

这看上去比较好，因此您通过把标准输出重定向到文件 **best-books** 中，来保存编号后的列表：

```
nl books > best-books
```

现在您已创建好所喜欢书的列表，并且还有行号，这可以给您的约会留下深刻的印象。

nl 程序是一个老程序了，可以追溯到 Unix 的初期。从传统上讲，**nl** 程序被用来在打印之前向文本中插入行号。例如，假设您有两个科学数据文件：**measurements1** 和 **measurements2**。您希望打印所有的数据，为了帮助解释数据，还希望打印输出的每一行数据都有编号。但是，您不希望改变原始的数据。

方法就是使用 **nl** 程序对行进行编号，然后再将输出重定向到 **lpr** 程序，后者将把数据发送给默认打印机(Unix 中打印文件的两个基本程序是 **lp** 和 **lpr**)。

```
nl measurements1 measurements2 | lpr
```

通过这种方式，所创建的临时编号只使用一次，然后就被抛弃。

在 Unix 初期，终端在纸张上打印输出，这通常很慢。后来，常见的做法就是将数据直接发送给真实的打印机，这样就比终端快许多。现在，通常是在屏幕上显示数据。在这种情况下，所需做的就是将 **nl** 的输出管道传送给 **less**(参见第 21 章)，而 **less** 将每次一屏地显示输出：

```
nl measurements1 measurements2 | less
```

再次说明，原始的数据并没有被改变。当使用 **nl** 时，行号总是临时的，除非将输出保存到文件中。

默认情况下，**nl** 按 1、2、3……的规律生成编号，这种方式不错。但是，如果需要，**nl** 还提供有几种选项，用来控制编号。使用 **-v** 选项可以改变起始编号，使用 **-i** 选项可以改变增量。下面举几个例子说明 **nl** 选项的使用方式，这个例子使用的文件是 **data**，该文件中包含几行文本。

第一个例子从 100 开始编号：

```
nl -v 100 data
```

输出为：

```
100 First line of text.
```

```
101 Second line of text.
102 Third line of text.
103 Fourth line of text.
```

接下来的例子从 1 开始编号(默认情况), 但是增量是 5:

```
nl -i 5 data
```

输出为:

```
1 First line of text.
6 Second line of text.
11 Third line of text.
16 Fourth line of text.
```

第三个例子使用两个选项, 从 100 开始编号, 增量是 5:

```
nl -i 5 -v 100 data
```

输出为:

```
100 First line of text.
105 Second line of text.
110 Third line of text.
115 Fourth line of text.
```

除了 **-v** 和 **-i** 外, **nl** 程序还有许多格式化选项。但是, 只有两个选项您可能经常需要。首先, 默认情况下, 如果数据中有空行, 那么 **nl** 将不对空行编号。为了强制 **nl** 对所有行编号, 可以使用 **-b**(body numbering, 正文编号)选项, 后面跟字母 **a**(all lines, 所有行):

```
nl -b a file
```

-b 选项还有其他变体, 但是很少使用。其次, 还可以使用 **-n**(number format, 数字格式)选项, 后面再跟一个代码来控制数字的格式:

ln=左对齐, 没有前导 0

rn=右对齐, 没有前导 0

rz=右对齐, 有前导 0

下面举一个例子:

```
nl -v 100 -i 5 -b a -n rz file
```

该命令生成从 100 开始的编号, 所使用的增量为 5。所有行都编号, 包括空行, 而且数字是右对齐的, 有前导 0。

18.2 统计行、单词和字符数量: **wc**

相关过滤器: **nl**

wc(word count, 单词统计)程序统计行、单词和字符的数量。所统计的数据可以来自另一个程序或者一个或多个文件。**wc** 程序的语法比较简单:

```
wc [-cLLw] [file...]
```

其中 *file* 是文件的名称。

wc 程序非常有用，实际上，它要比您刚开始意识到的更有用。这是因为可以在管道线中使用 **wc** 分析任何程序的文本输出。例如，假设您希望知道某个特定目录中有多少个文件。您可以手工统计所有的文件，或者生成一个列表，然后将列表管道传送给 **wc** 程序统计行数(稍后将示范一个例子)。

我们首先从基本知识入手。默认情况下，**wc** 的输出包含 3 个数字：数据中的行数、单词数和字符数。例如，**wc** 可能报告文件包含 2 行、13 个单词以及 71 个字符。

当输入来源于文件时，**wc** 将在 3 个数字之后写上文件名。如果指定不止一个文件，**wc** 将为每个文件显示一行输出，并且还有一个额外行显示总统计数——所有文件加在一起的行数、单词数和字符数。

下面举例说明。假设您在为自己的爱人写一首情人节的浪漫诗，下面就是您写的内容：

```
There was a young man from Nantucket,  
Whose girlfriend had told him to
```

为了统计诗中共有多少行、多少个单词以及多少个字符，可以使用：

```
wc poem
```

输出为：

```
2 13 71 poem
```

换句话说，该文件共有 2 行、13 个单词和 71 个字符。如果记不清哪个数字代表什么含义，则只需记住：行(Line)、单词(Word)、字符(Character)(如果您是男人，那么只需记住一个缩写词 LWC，即“Look at Women Carefully(小心地看女人)”)。

下面介绍技术细节：

- “字符”就是字母、数字、标点符号、空格、制表符或者新行字符。
- “单词”就是一串连续的字符，用空格、制表符或新行字符分隔。
- “行”就是以新行字符结尾的一串字符(新行字符在第 7 章中讨论过)。

正如前面所述，如果同时指定了不止一个文件，那么 **wc** 还显示统计总和。例如：

```
wc poem message story
```

下面是一些典型的输出*：

```
2 13 71 poem  
15 61 447 message  
43 552 3050 story  
60 626 3568 total
```

* 这些文件包含的是现实生活中的数据。poem 是我们前面使用的示例诗；message 来自编辑给我的一封电子邮件，日期是 2006 年 2 月 16 日，询问书什么时候完成；story 是“Late One Night”，是我写的，可以在我的网站 www.harley.com 上找到。

根据约定，输出总是按下列顺序显示：行数、单词数、字符数。如果不希望显示全部 3 个数字，则可以使用选项：**-l**(统计行)、**-w**(统计单词)、**-c**(统计字符)。当使用选项时，**wc** 只显示请求的数字。例如，为了只查看文件 **story** 的行数，可以使用：

```
wc -l story
```

输出为：

```
43 story
```

为了查看文件 **message** 中共有多少个单词和字符，可以使用：

```
wc -wc message
```

输出为：

```
61 447 message
```

-c(字符)、**-l**(行)和**-w**(单词)选项在 **wc** 命令中存在已经有数十年了，可以在任何类型的 Unix 和 Linux 系统中使用。对于 Linux 来说，还有另外一个选项：**-L**。该选项显示输入中最长行的长度。例如，假设您正在准备一个大的聚会，您希望给看门人一个不允许进入的人员的列表。您创建了一个文件 **do-not-admit**，该文件包含下述内容：

```
Britney
Paris
Nicole
Lindsay
```

为了显示该文件中最长行的长度，可以使用：

```
wc -L do-not-admit
```

在这个例子中，第 1 行和第 4 行都有 7 个字符，所以输出为：

```
7 do-not-admit
```

-L 选项在需要决定文件是否需要某些类型的格式化时派得上用场。例如，如果文件中有些行大于 70 个字符，那么可能在将文件发送给 **pr** 准备打印之前需要使用 **fmt** 格式化文本(本章后面解释)。

从经验可知，**wc** 主要有两种应用方式。首先，有时候需要快速测量文件的大小。例如，假设您通过电子邮件向某人发送一个文件。该文件非常重要，您希望确认这个文件完整到达。对原始文件运行 **wc** 命令，然后告诉接收方对接收到的文件也运行 **wc** 命令。如果两组结果匹配，就可以确信该文件完整无缺地到达。同样，如果您正在写一篇不少于 2000 字的论文，则可以时不时地使用 **wc -w** 命令查看您离目标还有多远。

wc 的第二种应用方式有所不同，但是同样重要：可以将某个命令的输出管道传送给 **wc**，以查看生成了多少行文本。因为许多程序生成的结果每行包含一项信息，所以可以通过统计行数，知道生成了多少信息。下面举两个例子。

第一个例子，**ls** 程序(参见第 24 章)列举目录中文件的名称。例如，下述命令显示/etc 目录中所有文件的名称(我们将在第 24 章中讨论目录)。

```
ls /etc
```

ls 程序有许多选项。但是，统计文件数量的选项并不存在。为了实现该目的，必须将 **ls** 的输出管道传送给 **wc**。因此，为了统计/etc 目录中文件的数量，可以使用：

```
ls /etc | wc -l
```

(大家可以在自己的系统上试一试。)

下面举第二个例子。在第 8 章中，示范了如何使用 **who** 命令查看登录到系统的用户标识。为了显示已经登录到系统的用户标识的数量^{*}，只需统计 **who** 命令输出的行数：

```
who | wc -l
```

如果您想更精巧一些，可以将最后一个管道线与 **echo** 程序(参见第 12 章)组合在一起，并通过命令替换(参见第 13 章)显示一个消息，说明当前有多少用户标识登录系统：

```
echo "There are `who | wc -l` userids logged in right now."
```

例如，如果有 5 人登录系统，将看到：

```
There are 5 userids logged in right now.
```

如果使用的是一个多用户系统，那么这是一条十分有趣的命令，可以放在登录文件(参见第 14 章)中。

18.3 Unix 使用制表符的方式

当查看键盘时，会发现键盘上有一个<Tab>键。这个键继承自打字机上所使用的制表符。尽管我们已经不再使用制表符，但是仍然使用<Tab>键，而且 Unix 仍然使用制表符设置。为了理解其原因，我们先返回到打字机主宰办公机器领域的时代。

单词“tab”是“tabulate，制表”的缩写，意味着将信息组织成表格。老式打字机上的<Tab>键被设计用来帮助按列排列信息，并在段落的开头缩进文本。下面举一个示范如何使用<Tab>键的例子。

假设您正在使用一台老式的打字机，您希望键入一个 3 列的表格。各列应该在位置 1、15 和 25 处对齐。在准备过程中，您分别在位置 15 和 25 处设置了两个小的机械标记，该标记称为制表位(tab stop)。

一旦完成这个之后，按下<Tab>将使托架水平移动到下一个制表位。例如，如果您在位置 8 上，那么按下<Tab>键托架将移动到位置 15。如果您在位置 19 上，那么按下<Tab>

^{*} 一个用户标识(userid，发音为“user-eye-dee”)并不是指一个人。它是用来登录 Unix 系统的名称。正如第 4 章中讨论的，Unix 只知道用户标识，不知道用户。

键托架将移动到位置 25。因此，以这种方式设置制表位可以方便地跳到位置 15 和 25 上，而不必重复地按<Space>键(而且还不必在不小心按<Space>键太多次的情况下返回)。

现在就准备好输入表格了。首先，您在打字机中插入一张纸，并将托架定位到行的开头。键入第一列的信息，然后按<Tab>键。这将使托架移动到位置 15。键入第二列的信息，然后再次按<Tab>键。现在托架将移动到位置 25。然后再键入第三列的信息。现在就结束了表格第一行的键入。向左推动托架返回控制杆，使托架移动到下一行的开头，从而准备好键入下一行信息。

尽管原始的 Unix 终端(参见第 7 章)并不是打字机，但是它们确实在纸张上打印信息，并且在遇到制表符时可以水平跳跃。基于这一原因，Unix 被设计为无论何时，当终端遇到制表符时，它都像打字机一样将光标移动到当前行的下一个制表位，而且时至今日，情况依然如此。Unix 终端通过将光标移动到下一个制表位“显示”制表符。

默认情况下，Unix 假定每隔 8 个字符(从位置 1 开始)有一个制表位。因此，默认 Unix 的制表位是 1、9、17、25、33 等。当键入文本并按<Tab>键时，Unix 就插入一个不可见的制表符。以后，当查看文本时，终端将通过创建足够多的水平空格跳到下一个制表位来“显示”制表符，就像打字机上的<Tab>键一样。

考虑下面的例子。您有一个一行的文件，该文件包含一个字母“A”、一个制表符、字符串“BBBBB”、另一个制表符以及字符串“CCC”：

```
A<Tab>BBBBB<Tab>CCC
```

如果使用 **cat** 命令显示该文件，将看到：

```
A      BBBBB   CCC
```

其中 **A** 位于位置 1 上。**BBBBB** 从位置 9 开始，而 **CCC** 从位置 17 开始。

当然，您看不到制表符：它们看上去就像真空区。因此，对于您的眼睛来说，字母之间的空隙就像空格字符一样，而空格字符同样是不可见的。例如，在上面的例子中，当查看 **cat** 命令的输出时，您无法辨别 **A** 和 **BBBBB**(在这个例子中)之间的真空区是 1 个制表符还是 7 个空格。

因此，这就出现了一个问题：当希望缩进文本或者按列对齐数据时，是使用制表符好呢还是使用空格好呢？程序员已经对这个问题争论了很长一段时间，因为他们需要使用真空区缩进控制流结构(if-then-else、while 循环等)。

一些程序员倾向于使用制表符进行缩进，因为它们比较简单。例如，每按一次<Tab>键，它就插入一个制表符，从而自动地将文本缩进到下一个制表位。如果使用的是空格，则需要按<Space>键多次，才能手工对齐文本。

另外，制表符也比空格灵活。例如，如果希望修改在屏幕上看到的缩进量，则只需在文本编辑器程序中改变制表位设置。如果使用的是空格，则不得不遍程序中的每一行，增加或者删除实际空格字符。

其他程序员则坚持使用空格进行缩进。他们说制表符很笨拙，因为它们生成的空格数量是变化的。他们指出，一个制表符可能表示 1 至 8 个空格，这取决于它在行中的位置。而当使用空格时，使用的空格数量正好是键入的空格数量：键入 4 个空格，得到的就是 4

个空格。

此外，尽管确实可以在大多数文本编辑器中调整制表位设置，但是大多数时间只能限于使用默认的制表位，即位置 1、9、17、25 等。这种间隔方式有时显得太多余，因为它创建了大缩进，使文本难以阅读。通过使用真实的空格，可以按照自己的意愿任意缩进 2、3 或者 4 个空格，完全满足自己的需要，而不管使用的是文本编辑器还是其他程序。

当然，生成水平间隔的需求不仅仅限于计算机程序。无论何时，当使用要求缩进或者按列组织的任意类型的文本时，必须选择使用制表符还是空格。

我无法告诉您使用哪一个，因为每个人都有自己的喜好，认为某一种更适应自己。随着时间的过去，您将会知道哪个选择更适合您自己。(就个人而言，相对于制表符，我更倾向于使用空格)*。我能够告诉您的就是，无论您最终选择了哪一种，有两个 Unix 程序可以使生活更简单(**expand** 和 **unexpand**)，稍后我们再对这两个程序展开讨论。但是，首先我们需要讨论一个更基本的问题。

18.4 可视化制表符和空格

当使用包含制表符和空格的文件时，会出现一个问题。既然制表符和空格都是不可见的，那么如何才能辨别它们在哪里呢？

当使用诸如 **expand** 和 **unexpand**(我们将在下两节中讨论)之类的程序时，这可能非常重要。**expand** 程序将制表符改变成空格，而 **unexpand** 程序将空格改变成制表符。如果看不到制表符和空格，那么怎样知道命令按希望执行呢？

最简单的方法就是在文本编辑器或者字处理程序中查看文件，同时打开一个选项，查看不可见的字符。选择有两种。

对于 **vi** 编辑器(参见第 22 章)来说，使用的命令是：

```
:set list
```

空格依旧不可见，但是制表符都将显示为 **^I**，即在 ASCII 码中表示制表符的控制字符。为了关闭该选项，可以使用：

```
:set nolist
```

如果您知道如何使用 **vi**，那么这是该问题的一个出色的解决办法：既快又简单。实际上，这就是我查看文件中的空白符的方式。

如果不会使用 **vi**，那么您可以使用 Nano 或者 Pico 编辑器，第 14 章中曾提及这些编辑器(它们基本上是相同的编辑器：Nano 是 GNU 版本的 Pico)。在 Nano/Pico 中，可以通过按 **<Esc>P**(也就是说，先按 **<Esc>** 键，然后再按 **<P>** 键)查看空格和制表符。这将打开“Whitespace display mode，空白符显示模式”。为了关闭该模式，只需再次按 **<Esc>P** 键。

但是，在使用 **<Esc>P** 键之前，必须在 Nano/Pico 初始化文件中添加下述行，Nano/Pico

* 当编写程序时，我缩进 4 个位置。当编写 HTML(Web 页面)时，我缩进 2 个位置。

的初始化文件分别为 `.nanorc` 和 `.picorc`(参见第 14 章):

```
set whitespace "xy"
```

其中 `x` 是希望表示一个制表符的字符, 而 `y` 是希望表示一个空格的字符。

例如, 如果希望将制表符显示为+(加号)字符, 将空格显示为|(竖线)字符, 则需要将下述语句放到 Nano/Pico 的初始化文件中:

```
set whitespace "+|"
```

通过使用 `vi` 或者 Nano/Pico 查看空格和制表符是不错的方法。然而, `vi` 非常复杂(从第 22 章可以看出), 因此需要花费很长的时间学习如何使用它。Nano/Pico 比较简单, 但是像 `vi` 一样, 它也需要时间学习。

那么在第 14 章中讨论的基于 GUI 的非常简单的编辑器(Kedit 和 Gedit)适合完成该任务吗? 正如前面所述, 这些编辑器使用简单。但是, 它们的功能不够强大。特别是它们不允许查看不可见的字符, 因此现在先忘记它们吧。

如果系统中有 Open Office, 即一组开放源代码的办公应用程序, 则可以使用另一种解决方法。Open Office 的字处理程序使文本中制表符和空格的查看变得简单。只需打开 View 菜单, 选择“Nonprinting Characters”即可。关闭这一特征也采用相同的方式。该方法既简单又易用。

除了在文本编辑器或者字处理程序中查看文件外, 还有一种间接检查 `expand` 和 `unexpand` 效果的方法。这种方法使用 `wc -c` 命令(本章前面讨论过)显示文件中字符的数量。

因为每个制表符都是一个单独的字符, 所以当使用 `expand` 将文件中的制表符改变成空格时, 文件中字符的数量将增加。同理, 当使用 `unexpand` 将文件中的空格改变成制表符时, 文件中字符的数量将减少。尽管使用 `wc -c` 不能显示不可见的字符, 但是它可以说明 `expand` 和 `unexpand` 是否工作正常。

18.5 将制表符转换成空格: `expand`

相关过滤器: `unexpand`

正如本章前面讨论的, 当需要缩进文本或者按列对齐数据时, 既可以使用制表符, 也可以使用空格。选择使用哪一种方式由自己决定。但是, 无论您选择使用哪一种, 有时候都会发现所使用的数据中有制表符, 而您需要将制表符转换为空格。同样, 有时候所使用的数据中有空格, 而您需要将空格转换为制表符。在这些情况下, 可以使用 `expand` 程序将制表符转换为空格, 或者使用 `unexpand` 程序将空格转换为制表符。我们首先讨论 `expand` 程序。

`expand` 程序的语法为:

```
expand [-i] [-t size | -t list] [file...]
```

其中 `size` 是固定宽度制表符的大小, `list` 是制表位列表, 而 `file` 是文件的名称。

expand 程序将输入文件中所有的制表符改变成空格，并且同时维持与原始文本相同的对齐方式。默认情况下，**expand** 使用 Unix 惯例，即将制表位设置为每 8 个位置一个：1、9、17、25、33 等。因此，输入中的每个制表符都将被输出中的 1 至 8 个空格替换(好好地想一想，直到理解它的含义)。

作为示例，请考虑下面的 **animals** 文件，该文件包含有分列组织的数据。当显示 **animals** 文件时，将看到下述内容：

```
kitten cat
puppy dog
lamb sheep
nerd programmer
```

可以看出，这个文件包含 4 行数据。而您看不见的就是每行包含的两个单词由一个制表符分隔开：

```
kitten<Tab>cat
puppy<Tab>dog
lamb<Tab>sheep
nerd<Tab>programmer
```

(<Tab>标记表示一个单独的制表符字符。)

下述命令将每个制表符转换成相应数量的空格，并将输出保存在 **animals-expanded** 文件中：

```
expand animals > animals-expanded
```

新的文件现在包含的内容如下所示：

```
kitten<Space><Space>cat
puppy<Space><Space><Space>dog
lamb<Space><Space><Space><Space>sheep
nerd<Space><Space><Space><Space>programmer
```

(<Space>标记表示一个单独的空格字符。)

如果使用 **cat** 或者 **less** 显示新文件，那么显示结果与原始文件相同。但是，如果使用文本编辑器或者字处理程序查看不可见的字符(正如上一节所述)，就可以看到所有的制表符都变成了空格。更具体地讲，在每一行中，**expand** 程序移除了所有的制表符，并且插入足够多的空格，从而使下一个单词可以从下一个制表位处开始，在这个例子中，就是位置 9。

正如前面所述，**expand** 使用 Unix 的默认设置，假定制表位之间有 8 个位置。使用 **-t(tab stop, 制表位)** 选项可以改变该设置。**-t** 选项有两种使用方式。第一种，如果所有的制表位都是相同的间距，则可以在 **-t** 之后跟一个数字。

例如，假设您有一个大文件 **data**，该文件中包含一些制表符。您希望将这些制表符转换为空格，并将输出保存在文件 **data-new** 中。但是，您希望将制表位设置为每隔 4 个字符一个，而不是每隔 8 个字符一个，也就是说您希望制表位在位置 1、5、9、13 等处。使用下述命令：

```
expand -t 4 data > data-new
```

通过使用这种表示法，我们可以说 **-t 8** 等价于 Unix 的默认制表位设置，即每隔 8 个字符设置一个制表位；**-t 4** 将每隔 4 个位置创建一个制表位。

-t 选项还有一种使用方式。如果希望制表位位于特定的位置上，则可以指定一串用逗号分开的多个数字。在该列表中，编号从 0 开始。也就是说，**0** 指的是行中的第一个位置；**1** 指的是行中的第二个位置；依此类推。例如，为了将制表位设置在位置 8、16、22 和 57 上，可以使用：

```
expand -t 7,15,21,56 data > data-new
```

最后，当希望转换制表符时，还有一种选项可以使用，但是只能在行的开头使用。在这种情况下，使用的就是 **-i**(initial, 初始)选项，例如：

```
expand -i -t 4 data > data-new
```

该命令转换制表符，但是只转换行开头的制表符。其他所有的制表符保持不变。在这个例子中，因为 **-t** 选项存在，所以制表位位于位置 1、5、9 等上。

提示

expand 程序对含有制表符的文本文件的预处理非常有用，特别是当其他程序期望在将这些文件发送给它们之前，文件按列准确对齐时。

例如，下述管道线将文件 **statistics** 中的所有制表符都替换为空格符，原文件每隔 4 个位置一个制表位。在制表符替换之后，抽取每行的前 15 个字符，并对结果进行排序：

```
expand -t 4 statistics | cut -c 1-15 | sort
```

18.6 将空格转换成制表符：unexpand

相关过滤器：**unexpand**

为了将空格转换成制表符，可以使用 **unexpand** 程序。**unexpand** 程序的语法为：

```
unexpand [-a] [-t size | -t list] [file...]
```

其中 *size* 是固定宽度制表符的大小，*list* 是制表位列表，而 *file* 是文件的名称。

unexpand 程序和预期的工作方式相同，就像 **expand** 程序的逆过程，将空格替换为制表符，同时保持原始文件中的对齐方式。

和 **expand** 程序一样，**unexpand** 程序的默认制表符设置也是隔 8 个位置，即 1、9、17 等。为了改变默认设置，可以使用与 **expand** 程序形式相同的 **-t** 选项：固定间隔(例如 **-t 4**，每隔 4 个位置)或者制表位列表(例如 **-t 7,15,21,56**)。如果使用的是列表，那么编号从 0 开始。也就是说，**0** 指的是行的第一个位置；**1** 指的是行的第二个位置；依此类推。

expand 和 **unexpand** 之间的一个重要区别就是默认情况下，**unexpand** 程序只替换行开头的空格。这是因为大多数时候，只使用 **unexpand** 程序进行缩进。一般情况下，不希

望替换行中间的所有空格。但是，如果希望改变默认设置，则可以使用**-a**(all, 全部)选项。这将告诉 **unexpand** 替换所有的空格，即使这些空格没有位于行的开头。

作为示例，假设您是著名的西海岸大学的一名学生，主修美国当代文化。您刚刚出席了一个有关米老鼠的研讨会，在研讨会上您认真地做了笔记。当您回家后，使用文本编辑器将笔记键入到文件 **rough-notes** 中。碰巧的是，当您缩进各行时，文本编辑器为每个缩进级别插入 4 个空格：

```
Mickey Mouse (1928-)
  Major figure in American culture
  Girlfriend is Minnie Mouse
    (Why same last name, if they are not married?)
  Did not speak until 9th film; The Karnival Kid, 1929
    First words were "Hot Dogs"
For exam, be sure I am able to:
Compare Mickey to President Roosevelt & Elvis Presley
Contrast Mickey with Hamlet (both tortured souls)
```

您希望将所有行开头的空格转换成制表符，并将数据保存到文件 **mickey** 中。使用的命令为：

```
unexpand -t 4 rough-notes > mickey
```

一旦运行这条命令，文件 **mickey** 中就包含：

```
Mickey Mouse (1928-)
<Tab>Major figure in American culture
<Tab>Girlfriend is Minnie Mouse
<Tab><Tab>(Why same last name, if they are not married?)
<Tab>Did not speak until 9th film; The Karnival Kid, 1929
<Tab>First words were "Hot Dogs"
For exam, be sure I am able to:
<Tab>Compare Mickey to President Roosevelt & Elvis Presley
<Tab>Contrast Mickey with Hamlet (both tortured souls?)
```

(<Tab>标记表示一个单独的制表符字符。)

18.7 格式化行：fold

相关过滤器：**fmt**、**pr**

在本节中，将讨论 3 个重新格式化文本的程序，从而使文本易于阅读，并且适于打印。**fold** 程序处理行，**fmt** 程序处理段落，而 **pr** 程序处理页面和列。我们首先从 **fold** 程序开始讨论。

fold 程序执行一项简单的任务：将长行分隔成短行。如果您有一个文件，这个文件中有过长的行，那么 **fold** 程序正好能派上用场。**fold** 程序可以根据指示瞬间分隔该行，如果

手工去完成的话，这个任务要花费大量的时间。**fold** 程序的语法为：

```
fold [-s] [-w width] [file...]
```

其中 *width* 是新行的最大宽度，*file* 是文件的名称。

默认情况下，**fold** 程序在位置 80 处分隔行。原因是：在 20 世纪 70 年代，当开发 Unix 时，人们认为 80 是关于文本行的一个约整数(参见下一节)。现在，大多数人不希望文本太宽，因此 80 通常太长。为了改变这个宽度，可以使用 **-w**(width, 宽度)选项，后面跟希望的最大行长度。例如，为了从文件 **long-lines** 中读取数据，将它重新格式化为每行不超过 40 个字符，并将输出保存在文件 **short-lines** 中，可以使用：

```
fold -w 40 long-lines > short-lines
```

当 **fold** 程序分隔行时，它所做的就是在合适位置上插入一个回车符(参见第 7 章)，使一行变成两行。下面是一个示范 **fold** 程序如何运转的例子。您有一个文件 **alphabet**，该文件有两行，如下所示：

```
abcdefghijklmnoprstuvwxyz  
ABCDEFGHIJKLMNopRSTUVWXYZ
```

每行有 26 个字符长。下述命令通过在第 13 个字符之后插入一个回车，将每行都分成两半：

```
fold -w 13 alphabet
```

输出为：

```
abcdefghijklm  
noprstuvwxyz  
ABCDEFGHIJKL  
MNopRSTUVW  
XYZ
```

当以这种方式使用时，**fold** 以最大长度分隔每个长行。例如，在上一个例子中，**fold** 将各个行正好分隔成 13 个字符。当数据经过这种改变之后没有变形时，这没有问题。但是，大多数时候，希望重新格式化的行包含的是单词，而对于这种类型的文本来说，人们不希望 **fold** 从单词的中间分隔行。

例如，许多字处理程序将每个段落存储为一个单独的长行。当文本以这种方式存储时，分隔只位于段落之间。假定您从一个字处理程序中复制了下述文本，则您在下面看到的内容实际上位于一个很长的行：

```
"Man cannot survive except through his mind. But the mind  
is an attribute of the individual. There is no such thing  
as a collective brain. The man who thinks must think and  
act on his own."
```

您希望将这个段落格式化为每行 40 个字符。为此可以使用命令：

```
fold -w 40 speech
```

但是，**fold** 正好在第 40 个字符处分隔行，这意味着一些行是在单词的中间分隔开的：

```
"Man cannot survive except through his m
ind. But the mind is an attribute of th
e individual. There is no such thing as
a collective brain. The man who thinks
must think and act on his own."
```

另一种方法就是使用 **-s** 选项，告诉 **fold** 不分隔单词：

```
fold -s -w 40 speech
```

输出为：

```
"Man cannot survive except through his
mind. But the mind is an attribute of
the individual. There is no such thing
as a collective brain. The man who
thinks must think and act on his own."
```

尽管右边的边缘有点参差不齐，但是单词保持了完整。如果希望保存格式化后的文件，只需重定向输出：

```
fold -s -w 40 speech > speech-formatted
```

提示

对于一些程序来说，可能发现每次使用这个程序时都使用相同的选项。为了使工作方便，可以定义一个包含选项的别名(参见第 13 章)，从而不用每次都要键入它们。

例如，假设您经常使用 **fold** 命令的 **-s -w 40** 选项，那么您可以在环境文件中放入下述别名定义(参见第 14 章)。其中第一个别名定义是针对 Bourne shell 家族的；第二个别名定义是针对 C-Shell 家族的。

```
alias fold="fold -s -w 40"
alias fold "fold -s -w 40"
```

现在，每次使用 **fold** 时，都可以自动地获得 **-s -w 40** 选项。

如果有时候希望运行没有 **-s -w 40** 选项的 **fold** 命令，那么您可以通过在命令名称前面键入一个 ****(反斜线)字符临时挂起别名。例如，如果您希望使用 **fold** 命令的 **-w 60** 选项，而不是 **-s -w 40** 选项，则可以使用命令：

```
\fold -w 60 long-text > short-text
```

为了运行没有选项的 **fold** 命令，可以使用：

```
\fold long-text > short-text
```

18.8 80 字符行

多年以来,程序员在文本中一直应用每行 80 个字符,而且终端在输出中也显示每行 80 个字符。在 GUI 出现之后,它允许动态地调整窗口的大小,魔幻数字 80 已经不再是行长度的准确数字。然而,许多 Unix 程序仍然使用 80 字符/行作为默认设置,例如:

- **fold** 程序(上一节中讨论过)默认情况下在位置 80 处分隔行。
- 如果查看 Unix 联机手册(参见第 9 章)中的页面,会发现它们也被格式化为每行 80 个字符。
- 当使用终端仿真器程序(参见第 3 章)时,默认行宽通常也是 80 字符。

为什么是这种情况呢? 80 字符/行有什么特殊含义吗? 下面就是这个问题的故事。

1879 年,美国发明家 Herman Hollerith(1860-1929)为即将到来的 1880 年人口普查设计一种处理信息的系统。他从纺织工业中借鉴了一种思想,即从 19 世纪初期开始,纺织工业使用上面有洞的大型卡片控制自动织布机这一思想。Hollerith 采纳了这一思想,并开发了一种系统,在该系统中,人口普查数据存储在穿孔卡片上,每个人一张卡片。Hollerith 所设计的卡片与美国的钞票大小相同*,从而允许他使用现有的货币设备——例如装填接收器(filling bin),来处理卡片。卡片通称为穿孔卡片,有 20 列,后来扩展到 45 列。

Hollerith 的系统证实非常有用,1896 年,他创立了制表机器公司(Tabulating Machine Company, TMC),制造他自己的机器。1911 年, TMC 公司合并了两家其他公司——美国计算尺公司(Computing Scale Company of America)和国际时间记录公司(International Time Recording Company),并且成立了计算制表记录公司(Computing Tabulating Recording Company, CTR)。除了制造制表器和穿孔卡片外,CTR 还制造商业标尺、工业时间记录器以及肉和奶酪切片机。1924 年,CTR 公司正式将名称改成国际商用机器公司(International Business Machines, IBM)。

到 1929 年,IBM 公司的技术已经发展到能够增加穿孔卡片的列数。使用这种新技术,IBM 公司的穿孔卡片——也就是旧美元钞票大小——已经大得足够存放 80 列,每列可以存储一个字符。因此,当 20 世纪 50 年代末期第一台 IBM 计算机开发出来时,他们使用了能够存储 80 个字符的穿孔卡片。结果,程序和数据都存储为每行 80 个字符,而且在极短的时间内,这成为事实上的标准。

到 20 世纪 80 年代,随着程序员开始使用终端,以及后来个人计算机的应用,穿孔卡片被逐步淘汰。但是,80 字符标准被坚持下来,终端和 PC 使用的屏幕每行都显示 80 个字符,这也是程序员(和程序)所期望的。而 Unix 也正是在这个时代开发出来的,因此每行只能是 80 个字符,这样才能融入 Unix 文化中去。

现在您应理解为什么过了 25 年,而且 GUI 如此流行,但每行 80 个字符仍然存活在 Unix 世界的各种隐蔽角落中的原因了。

* 1862 年,美国政府发行了第一张钞票,一个 1 美元的钞票,大小为 $7\frac{7}{8}$ 英寸 \times $3\frac{1}{8}$ 英寸。这就是 Hollerith 时代的钞票大小,因此,也是他的穿孔卡片的大小。1929 年,美国政府缩减了 20% 的钞票尺寸,使钞票大小变为 $6\frac{1}{8}$ 英寸 \times $2\frac{3}{8}$ 英寸,这就是现在钞票的大小。

18.9 格式化段落：fmt

相关过滤器：**fold**、**pr**

fmt 程序格式化段落。**fmt** 的目标就是将段落中的各行连接在一起，从而使段落尽可能短小和紧凑，而且还不改变内容和空白符。换句话说，就是 **fmt** 使文本看上去更漂亮。

fmt 程序的语法为：

```
fmt [-su] [-w width] [file...]
```

其中 *width* 是行的最大宽度，*file* 是文件的名称。

当 **fmt** 读取文本时，它假定段落由空行分隔。因此，一个“段落”就是一个或者多个连续的文本行，不包含空行。**fmt** 程序根据下述规则每次读取并格式化一个段落。

- 行宽：使每行尽可能的长，但是不能超过指定的长度。默认情况下，最大行宽是 75 个字符，但是可以使用 **-w** 选项改变行宽。在改变行宽时，只需在 **-w** 后面跟着希望的行宽即可。例如 **-w 50**。
- 句子：无论何时，尽可能地在句子末尾分隔行。避免在句子的第一个单词之后或者最后一个单词之前分隔行。
- 空白符：保持单词以及空行之间的所有缩进、空格。使用 **-u** 选项(参见下面)可以修改这种情况。
- 制表符：在读取文件时将所有的制表符转换成空格，并且在最后的输出中的合适位置上插入新的制表符。

作为示例，假设您有一个文件 **secret-raw**，该文件包含下述 3 个文本段落。注意各行并没有均匀地格式化：

```
As we all know, real
success comes slowly and
is due to a number of different factors all coming
together
over a period of years.
```

```
Although there
is no real shortcut, there is a secret: a secret so
powerful that you can use it to
open doors that might otherwise be closed, and to
influence
people to help you time and again. In fact, I would
go as far as to say that this is the secret that has
a lot to do with my success.
```

```
The secret is simple...
```

您希望使用行宽 50 字符对该文本进行格式化，并将结果保存在文件 **secret-formatted** 中。完成这一任务的命令为：

```
fmt -w 50 secret-raw > secret-formatted
```

secret-formatted 文件中现在包含的内容如下所示:

```
As we all know, real success comes slowly and is
due to a number of different factors all coming
together over a period of years.
```

```
Although there is no real shortcut, there is a
secret: a secret so powerful that you can use it
to open doors that might otherwise be closed,
and to influence people to help you time and
again. In fact, I would go as far as to say
that this is the secret that has a lot to do
with my success.
```

The secret is simple... *

fmt 程序还有其他几个选项,但是只有两个选项比较重要。**-u**(uniform spacing, 统一间距)选项告诉 **fmt** 减少空格,从而使单词之间最多只有一个空格,而且句子末尾最多只有两个空格,这一样式称为法国式间距(French spacing)**。例如,文件 **joke** 中包含下述文本:

```
A man walks      into a drug store and goes up to      the
pharmacist. "Do you sell talcum powder?" asks
the      man. "Certainly," says
the pharmacist, "just walk
this way." "If I could walk that way," says the
man,      "I wouldn't need talcum      powder."
```

使用下述命令格式化上述文本:

```
fmt -u -w 50 joke
```

输出为:

```
A man walks into a drug store and goes up to the
pharmacist. "Do you sell talcum powder?" asks
the man. "Certainly," says the pharmacist, "just
walk this way." "If I could walk that way,"
```

* 这个例子取自论文“The Secret of My Success”。如果您希望阅读整篇论文,可以在我的官方网站 www.harley.com 中查找。

** 对于法国式间距来说,句子后面是 2 个空格,而不是 1 个空格。这种样式通常在等宽字体中使用,在等宽字体中所有的字符都是相同宽度。对于这样的字体来说,在句子末尾有一个额外的空格将方便眼睛的观察。

两个主要的 Unix 文本编辑器 **vi** 和 **Emacs** 都认可法国式间距,因此能够检测句子的开头和结尾。这就允许 **vi** 和 **Emacs** 将句子作为完整的单元进行处理。例如,您可以删除两个句子、改变一个句子、向后跳三个句子,等等。基于这一原因,许多 Unix 人士形成了在句子后面使用两个空格的习惯(我也是这样,甚至在键入电子邮件时也是)。


```
says the man, "I wouldn't need talcum powder."
```

注意第一个句子的末尾只有一个空格。这是因为原始文件中只有一个空格，而 **fmt** 不增加空格，它只移除空格。

最后一个选项是 **-s**(split only, 仅拆分)，该选项告诉 **fmt** 拆分长行，但是不连接短行。当处理希望保存原格式的文件(例如编写计算机程序)时可以使用这个选项。

18.10 打印的旧时代

在接下来的两节中，我们准备讨论 **pr** 程序，**pr** 是 Unix 初期创建的一个程序，即 20 世纪 70 年代。那个时代，打印机非常昂贵，没有人拥有自己的打印机，因此设计 **pr** 是用来在共享环境中为打印准备文件的。但是，**pr** 程序除了基本的功能——打印功能之外，在格式化文本方面也有自己的用途。

不过，在讨论这些话题之前，我希望先铺垫一些基础知识，描述一下 Unix 初期是如何打印文件的。

因为打印机非常昂贵，所以打印机总是由一组人共享。无论何时，当用户希望打印文件时，他需要在自己的终端上输入合适的命令，以格式化并打印文件。或者他运行一个生成可打印输出的程序。每个打印请求都称为一个“打印作业”，一旦生成一个打印作业，它就被放在“打印队列”中等待轮到它打印。通过这种方式，一个接一个的打印作业被生成、存储，最后被打印。

实际打印机可能位于计算机室，即由许多人(通常是程序员)使用的公共区域。输出被打印在连续的、折叠的计算机打印纸上，单个打印作业的输出称为一个“打印输出(printout)”。随着打印输出的积累，有人——通常是“操作员”，在计算机室中工作——在穿孔处将纸张撕开，从而分隔打印输出。然后他将每个打印输出都放在一个箱子中，发起打印请求的人接下来就可以从这个箱子中取走他打印的东西。

因为系统的组织方式如此，所以必须有一种方式，使操作员能够接收一堆打印纸，并将它分成单独的打印作业。**pr** 程序就是满足该需求的，它通过提供两种服务既满足用户的需求，也满足操作员的需求。首先，**pr** 将文本格式化成页；其次，**pr** 确保每页都有自己的标题、边缘和页号。通过这种方式，打印输出不仅看起来漂亮(对于用户来说)，而且还易于组织(对于操作员来说)。

现在，大多数人认为 **pr** 程序只用于打印，这是错误的。确实，**pr** 仍然能够完成它的设计目的：准备发送给打印机的输出(因此命名为 **pr**)。这仍然是一个有用的功能，因此我们将在下一节中讨论 **pr** 的这些方面。

但是，**pr** 程序不仅仅是将文本简单地分成页并生成标题、边缘和行号，它还可以完成许多工作。它可以以几种非常有用的方式格式化文本，特别是在将 **pr** 和 **fold** 与 **fmt** 组合在一起使用时。例如，您可以使用 **pr** 将一个单独文件中的文本按列排列；还可以将多个文件中的文本合并在一起，每个文件进入自己的列。因此，一旦讨论完 **pr** 的基本功能，我们将示范如何以与打印无关的方式使用 **pr**，所有这些方式都非常高效和聪明。

提示

现在，大多数人不再使用基于文本的工具来打印普通文件，而是使用图形工具，例如字处理程序，以便使格式和分页的控制更加简单。

但是，如果您是一名程序员，那么您将会发现，当有需要时，传统的 Unix 工具(**pr**、**fmt**、**nl**、**fold**)在打印源代码方面还是相当出色。

18.11 按页格式化文本：pr

相关过滤器：**fold**、**fmt**

pr 的主要功能是按页格式化文本，以使其适合于打印。**pr** 程序还可以按列格式化文本，并且还可以合并多个文件中的文本，我们将在下一节中讨论这些内容。**pr** 程序的基本语法如下所示：

```
pr [-dt] [+beg[:end]] [-h text] [-l n] [-o margin] [-W width] [file...]
```

其中 *beg* 是需要格式化的第一页，*end* 是需要格式化的最后一页；*text* 是标题中间的文本；*n* 是每页的行数；*margin* 是左边缘的大小；*width* 是输出的宽度；*file* 是文件的名称。

通常将 **pr** 作为管道线的一部分，以便在将文本发送给打印机之前对文本进行格式化。例如，假设您有一个程序 **calculate**，您希望打印该程序生成的数据。下述管道线将 **calculate** 的输出发送给 **pr** 进行格式化，然后再发送给 **lpr** 进行打印(Unix 中文件打印的两个基本程序是 **lp** 和 **lpr**)。

```
calculate | pr | lpr
```

下面是一个类似的例子。这个例子将 3 个文件的内容组合、格式化并打印：

```
cat data1 data2 data3 | pr | lpr
```

默认情况下，**pr** 通过在顶端插入一个标题、左边插入一个边缘、底部插入一个页尾来格式化页面。标题和页尾各占用 5 行。左边缘和页尾只用于间距，所以它们是空白的。但是，标题在中间行上包含信息：文件上一次修改的日期和时间、文件的名称以及页号(根据所使用 **pr** 程序的版本不同，这些细节可能有些微变化)。作为示例，下面示范一个典型的标题。省略掉空白行，如果格式化一个文件 **logfile**，那么看到的标题如下所示：

```
2008-12-21 10:30          logfile          Page 1
```

pr 程序假定一页有 66 行。这是因为老式打印机使用 11 英寸的纸，每英寸打印 6 行。标题(在顶部)和页尾(在底部)各占用 5 行，则每页只剩下 56 行用于单倍行距的文本。当 **pr** 创建页时，它处理页 1、页 2、页 3，等等，直到所有数据都格式化完。

如果希望测试 **pr**，查看它的工作方式，那么一种简单的方法就是格式化一个文件，并将输出发送给 **less**(参见第 21 章)。这将允许您每次一屏地查看格式化的输出。例如，假设您参加了一门课程，撰写了一篇论文，这个论文存储在文件 **essay** 中。如果希望查看 **pr** 如何格式化该文本，可以使用：

```
pr essay | less
```

如果您对看到的结果满意，则可以将它发送给打印机：

```
pr essay | lpr
```

或者保存到一个文件中：

```
pr essay > essay-formatted
```

如果您的论文最初是用字处理程序写的，那么这篇论文中就有非常长的行。这是因为字处理程序将每个段落存储为一个长行^{*}。在这种情况下，需要首先使用 **fold -s** 或者 **fmt** 恰当地分隔行，这两个命令都特别适合该文本：

```
fmt essay | pr | less
fold -s essay | pr | less
```

几乎所有时候，当使用 **pr** 格式化页并进行打印时，默认设置已能满足需求。但是，如果有需要，也可以使用几个选项修改设置。最常用的选项就是 **-d**，该选项告诉 **pr** 使用双倍行距文本。

考虑另一个例子，这个例子格式化并打印文本文件 **essay**。这个简单的管道线首先使用 **fmt** 格式化文件的各行。**fmt** 的输出被发送给 **pr**，**pr** 将数据格式化为双倍行距文本的页面。最后，将 **pr** 的输出发送给打印机：

```
fmt essay | pr -d | lpr
```

结果是一个漂亮的、双倍行距并且是印刷格式的论文，该结果适合于编辑或者提供给老师。注意 **-d** 并不修改原始文本：它所做的就是确定按页格式化文本时所使用的间距类型。在这个例子中，原始文件 **essay** 保持不变。

如果希望控制格式化哪些页，可以使用语法：

```
pr +begin[:end]
```

其中 *begin* 是要格式化的第一页，*end* 是要格式化的最后一页。

例如，为了跳过第 1 页和第 2 页，也就是说，从第 3 页开始，直至文件的末尾，可以使用：

```
fmt essay | pr -d +3 | lpr
```

只格式化并打印第 3 页至第 6 页，可以使用：

```
fmt essay | pr -d +3:6 | lpr
```

如果希望指定标题中间部分的文本，可以使用 **-h** 选项，例如：

```
fmt essay | pr -h "My Essay by Harley" | lpr
```

为了改变每页的总行数，可以使用 **-l**，后面跟一个数字。例如，假设您只希望每页有 40 行文本，加上标题(5 行)和页尾(也是 5 行)，则需要每页有 50 行：

^{*} 字处理程序文档以一种特定的二进制格式存储。但是，几乎所有的 Unix 过滤器都假定数据以文本形式存储。因此，如果希望使用本章中的 Unix 程序来处理字处理程序文档的话，必须首先将字处理程序中的该文档保存成纯文本形式。例如，在上面的例子中，文件 **essay** 就是字处理软件文档 **essay.doc** 的纯文本形式的文件。

```
fmt essay | pr -l 50 | lpr
```

为了消除标题，可以使用 **-t** 选项。当使用 **-t** 时，页之间没有分隔，这意味着所有行都用于正文。当格式化不准备打印的文本时，这比较有用。例如，您可能希望将单倍行距文本修改为双倍行距文本。下述命令将把文件 **essay** 的内容格式化为双倍行距、没有标题，并且将输出保存到文件 **essay-double-spaced** 中：

```
fmt essay | pr -t -d > essay-double-spaced
```

默认情况下，**pr** 不插入左边缘。这样就可以行了，因为极有可能是打印机被设置成自动创建边缘。但是，如果希望自己添加一个额外的边缘，则可以使用 **-o**(offset, 偏移)选项，后面跟额外边缘的空格数量。另外，还可以使用 **-W** 选项(注意是大写字母 **W**)改变输出的宽度(默认值是 72 字符)。当使用 **-W** 选项时，太长的行会被截断，因此一定要小心不要丢失文本。下面示范一个特别有用的例子，描述如何使用这两个选项。

如果您是一名学生，那么您知道有时候要求的论文可能会比较长。例如，您写了一篇 8 页的论文，但是您的老师要求写一篇 10 页的论文。当然，您可以重新整理笔记、进行进一步的研究并重新撰写论文。或者，您可以加宽边距并采用双倍行距打印论文。^{*}

下述例子使用双倍行距设置格式化文件 **essay** 的内容，每行有 50 个字符，左边距是 5 个空格；也就是说，每行文本只有 45 个字符。结果就是使论文看上去比真实的长了许多：

```
fmt -w 45 essay | pr -d -o 5 -W 50 | lpr
```

如果在打印之前希望查看输出，则可以使用：

```
fmt -w 45 essay | pr -d -o 5 -W 50 | less
```

注意：**fmt** 使用 **-w 45** 选项非常必要，因为默认情况下，**fmt** 生成的行有 72 个字符长。但是，在这个例子中，我们要求 **pr** 将每行文本的输出限制为 45 个字符。因此，需要确保任何一行都不超过 45 个字符：否则，它们将被截断。作为 **fmt** 的备选，还可以使用 **fold -s** 程序，该程序生成相似的格式：

```
fold -s -w 45 essay | pr -d -o 5 -W 50 | lpr
```

提示

fold 和 **fmt** 都可以用于格式化文本行。那么应该使用哪一个程序呢？

fold 程序只完成一件事情：分隔行。默认情况下，**fold** 在指定列分隔行。但是，如果使用了 **-s** 选项，那么 **fold** 将在单词之间分隔行。这将使文本的右边多少有点混乱，但是保持了单词的完整性。

fmt 程序格式化段落。像 **fold** 一样，**fmt** 也分隔长行。但是，与 **fold** 不同，**fmt** 还连接短行。

因此，如果需要在特定点分隔行，则可以使用 **fold**。如果需要格式化包含短行的文本，则可以使用 **fmt**。这样就比较清楚了。

但是，如果需要在单词边界分隔行并且文本早已格式化，应该怎么办呢？在这种情况下，可以使用 **fold -s** 或者 **fmt**。注意，这两条命令有时候生成稍微有所不同的输出，因此可以分别试一试这两条命令，看看哪条命令最适合。

^{*} 别告诉您的老师这是我教的。

提示

将文本发送给 **pr** 之前使用 **fold** 或者 **fmt** 对文本进行预处理时，最好是明确指定希望的准确行宽，因为这 3 个程序使用不同的默认值：

- **fold**: 80 字符/行
- **fmt**: 75 字符/行
- **pr**: 72 字符/行

18.12 按列格式化文本：pr

正如上一节讨论的，**pr** 的目的就是将文本格式化为适合打印的页。除了前面讨论的简单格式化外，**pr** 还可以按列格式化文本。输入数据来源于单个文件或者几个文件。

当使用 **pr** 创建数据列时，**pr** 程序的语法为：

```
pr [-mt] [-columns] [-l lines] [-W width] [file...]
```

其中 *column* 是输出列的数量，*lines* 是每页的行数，*width* 是输出的宽度，*file* 是文件的名称。

我们首先从一个文件开始。为了指定输出列的数量，我们使用一个-(连字符)字符，后面跟一个数字。例如，为了分成两列，我们使用 **-2**。为了控制列的宽度，可以使用 **-l** 选项。下面是一个典型例子。

您是一家小规模，但是很著名的文艺类学校的本科生。您的学术指导老师认为，如果参加课外的活动，那么您获得医学院认可的机会将提高很多，因此您参加了反蛋黄酱运动。您的部分职责就是从事时事通讯。您刚刚完成了一篇有关为什么蛋黄酱不好的文章。您以纯文本形式保存了这篇文章，这个文件就是 **article**。

在将这篇文章导入到时事通讯程序中前，您需要将文本分成两列，并且每页的长度是 48 行。下述管道线完成这一任务。注意，在将文本发送给 **pr** 之前，**fmt** 程序将文本格式化为每行 35 个字符：

```
fmt -w 35 article | pr -2 -l 48 > article-columns
```

所需的格式化文本现在就存储在文件 **article-columns** 中。

那么数字 35 是如何得来呢？当使用 **pr** 程序时，默认行宽是 72。共有两列，每列的末尾至少有一个空格，这就剩下最多 70 个文本字符。将这个数除以 2 得到每列最多 35 个字符(如果使用 **-W** 选项改变了行宽，则必须相应地修改该计算过程)。

提示

将文本分列时，**pr** 会盲目地截断太长的行。因此，如果文本所包含行的长度大于列宽，那么在将文本发送给 **pr** 之前必须使用 **fold -s** 或者 **fmt** 分隔行。

默认情况下，**pr** 使用制表符，而不是空格对齐各列。如果希望使用空格，那么您需要做的就是使用 **expand**(本章前面讨论过)将制表符转换成空格。例如，下述命令行就在保存

数据之前将 **pr** 的输出管道传送给 **expand**:

```
fmt -w 35 article | pr -2 -l 40 | expand > article-columns
```

如果仔细地检查该命令的输出, 就会发现对齐方式是使用空格而不是制表符保持的(有关文本中可视化制表符和空格的信息, 请参见本章前面的讨论)。

pr 的最后应用就是将多个文件分别格式化成为单独的列。使用 **-m**(merge, 合并)选项, **pr** 将在单独的列中输出每个文件。例如, 为了将 3 个文件格式化成 3 个单独的列, 可以使用:

```
pr -m file1 file2 file3
```

当以这种方式格式化 3 个文件时, 每列的最大宽度默认情况下是 23 个字符^{*}。如果输入文件包含的行大于列宽, **pr** 就将行截断。为了避免发生这种情况, 在将文本发送给 **pr** 之前必须格式化文本。下面举一个例子。

您在准备一篇时事通讯, 并写了 3 篇新闻故事, 分别保存在文件 **n1**、**n2** 和 **n3** 中。您希望使用 **pr** 将这 3 篇故事分成一个页面上的 3 列, 每列一个故事。在使用 **pr** 之前, 必须使用 **fold -s** 或者 **fmt** 格式化文本, 以保证没有行超过 23 个字符。下述命令就完成这一任务, 并分别将输出保存在 3 个文件 **f1**、**f2** 和 **f3** 中:

```
fmt -w 23 n1 > f1
```

```
fmt -w 23 n2 > f2
```

```
fmt -w 23 n3 > f3
```

现在您可以使用 **pr** 格式化 3 篇文章了, 每篇文章都位于自己的列中:

```
pr -m f1 f2 f3 > formatted-articles
```

当通过这种方式合并多个文件时, 通常使用 **-t** 选项移除标题:

```
pr -mt f1 f2 f3 > formatted-articles
```

这将生成一个长的、连续的文本列, 而且没有中断。

18.13 练习

1. 复习题

1. **wc** 程序的 3 个基本选项分别是什么? 每个选项都是做什么用的? 什么使 **wc** 成为一个如此有用的工具? 当使用 **wc** 时, “行” 的定义是什么?
2. 当在 Unix 中使用制表符时, 默认制表符位置是什么?
3. **fold**、**fmt** 和 **pr** 程序都可以用于重新格式化文本。这些程序之间的主要区别是什么?
4. **fold**、**fmt** 和 **pr** 程序拥有不同的默认行长度。它们各是多少?

^{*} 默认情况下行宽是 72 个字符。在每列的末尾, 应该至少有一个空格。因为我们创建了 3 列, 从 72 中减去 3, 可以知道文本每行最多有 69 个字符。再除以 3, 得到每列最多有 23 个字符。

2. 应用题

1. 使用命令 `less /etc/passwd` 查看系统上的口令文件。注意该文件中每个用户标识对应一行信息，每行包含几个字段，字段之间用:字符分隔。第一个字段就是用户标识。创建一个管道线，生成一个包含系统中所有用户标识的有序且编号的列表。提示：先使用 `cut`(参见第17章)，接着使用 `sort`，然后使用 `nl`。

2. 命令 `ls` 显示工作目录中所有文件的列表(除了点文件)。创建一个管道线，统计文件的数量。提示：先使用 `ls`，然后使用 `wc` 及合适的选项。接下来，创建一条命令，像下述形式一样显示输出(其中 `xx` 是文件的数量)：

```
I have xx files in my working directory.
```

提示：使用 `echo`(第12章)和命令替换(第13章)，并利用刚才创建的管道线。

3. 到自己选定的一个网站上，复制一些文本到剪贴板中。在 Unix 的命令行上，使用 `cat` 程序创建一个文件 `webtext`：

```
cat > webtext
```

粘贴文本并按[^]D 键(第6章中讨论过复制和粘贴)。现在就拥有了一个包含网站中文本的文件。创建一个管道线，将该文本格式化每行 40 个字符，并将多个连续的空格转换成一个单独的空格。在管道线的末尾，每次一屏地显示文本。

4. 使用上个练习题中的文件 `webtext`，将文本分成适合于打印的两列页面。每列有 30 个字符宽，页面有 20 行长。所创建的各列应该包含空格，而不是制表符。一次一屏地显示格式化后的输出。一旦认为输出格式是正确的，就可以将输出保存在文件 `columns` 中。

3. 思考题

1. 正如本章所讨论的，80 列的行在计算世界广泛使用，这是因为在 20 世纪 50 年代，第一台 IBM 计算机使用的是穿孔卡片，每卡片只能存放 80 个字符。数字 80 只是一个偶然数字，因为穿孔卡片的大小取自美元钞票的大小。这就是旧技术如何影响新技术的一个例子。

与此相似，当 IBM 公司在 1981 年开发出 PC 时，键盘设计基于标准的打字机，打字机使用的就是所谓的标准打字机键盘(QWERTY)布局(根据最顶端左边的 6 个键命名)。人们普遍认为标准打字机键盘是一个很差的键盘，因为最重要的键位于特别难使用的位置上。尽管已经有更好的键盘布局，但是标准打字机键盘仍然是标准。您认为旧技术为什么对新技术有如此强的影响？这样有什么坏处？又有什么好处？(当您有空时，可以在 Internet 上查找一下 Dvorak 布局。我使用这种键盘已经有几年时间了，而且再也不希望换回去了。)

过滤器：选取、排序、组合及变换

在本章中，我们通过讨论 Unix 工具箱中最有趣最强大的过滤器来结束对过滤器的讨论，具体包括：选取数据、排序数据、组合数据以及变换数据的工具。这些程序非常有用，值得我们花时间详细讨论。

众所周知，功能强大的程序学习的时间也长，对于本章中讨论的过滤器来说也确实如此。实际上，这些程序功能如此强大，您可能永远都不能完全掌握全部的细枝末节。

在本章中我将确保您理解基本的知识，并且还会示范许多的例子加以说明。随着时间的流逝，以及您的技能和需求的发展，您不仅可以查看联机手册，以了解更高级的细节，而且还可以使用 Web 和 Usenet 向其他人寻求帮助。最重要的是，无论何时，如果有机会与 Unix 极客交谈，一定要请求他们示范使用这些过滤器的特别技巧。这是学习 Unix 的最佳方式。

本章是讨论过滤器的 4 章(第 16~19 章)中的最后一章。在第 20 章中，我们将讨论正则表达式，它一般用来指定模式。正则表达式可以增强过滤器的功能，而且在第 20 章中，我们将示范许多属于本章中过滤器的例子，特别是 **grep**，或许它就是这些过滤器中最重要的过滤器。

19.1 选取包含特定模式的行：grep

相关过滤器：look、strings

grep 程序从标准输入或者文件中读取数据，抽取所有包含特定模式的行，并写到标准输出。例如，可以使用 **grep** 搜索 10 个长文件，查找所有包含字符串“Harley”的行。或者，使用 **sort** 程序(本章后面讨论)对大量的数据排序，然后将数据管道传送给 **grep**，抽取所有以单词“note”打头的行。

除了搜索特定的字符串之外，还可以对 **grep** 使用正则表达式，以搜索模式。当这样做时，**grep** 将成为一个功能非常强大的工具。实际上，正则表达式非常重要，所以我们将第 20 章中专门讨论正则表达式，第 20 章还示范了许多使用 **grep** 的例子(实际上，稍后您就会明白，名称 **grep** 中的 **re** 代表“regular expression，正则表达式”)。

grep 程序的语法为：

```
grep [-cIlLnrsvw] pattern [file...]
```

其中 *pattern* 是要搜索的模式，*file* 是输入文件的名称。

下面举一个简单的例子。大多数 Unix 系统将每个用户标识的基本信息保存在一个命名为 **/etc/passwd** 的文件中(参见第 11 章)。每个用户标识对应一行信息。为了显示某用户标识的信息，可以使用 **grep** 在这个文件中搜索该模式。例如，为了显示用户标识 **harley** 的有关信息，可以使用命令：

```
grep harley /etc/passwd
```

如果 **grep** 没有查找到与指定模式匹配的行，则没有任何输出或者警告消息产生。与大多数 Unix 命令相似，**grep** 程序也简单扼要。当没有事情可说时，就不说任何事情(如果您认识的每个人都拥有这样的原则，那岂不是很好?)

当指定包含标点符号或者特殊字符的模式时，应该将它们引用，从而使 shell 能够正确地解释命令(有关引用的讨论请参见第 13 章)。例如，如果要在文件 **info** 中搜索冒号后跟一个空格的模式，可以使用命令：

```
grep ':' ' info
```

名称含义

grep

在 20 世纪 70 年代初期，最初版本的 Unix 所使用的文本编辑器就是 **ed**。在 **ed** 中，有一个命令可以用来在文件中搜索所有包含一个指定模式的行，然后在终端上打印这些行(那个时代，Unix 用户使用在纸张上打印输出的终端)。

这个命令就是 **g**，代表 **global**(全局)，因为它能够搜索整个文件。当使用 **g** 打印包含某个模式的所有行时，语法为：

```
g/re/p
```

其中 **g** 代表“**global**，全局”，**re** 就是描述希望搜索的模式正则表达式，**p** 代表“**print**，打印”。

grep 的名称就是来自这个偶然产生的缩写。换句话说，就是 **grep** 代表：

- **Global**(全局)：表明 **grep** 搜索全部输入数据。
- **Regular Expression**(正则表达式)：说明 **grep** 可以搜索能够用正则表达式(第 20 章中讨论)表示的任何模式。
- **Print**(打印)：一旦 **grep** 查找到所需的东西，**grep** 就打印(显示)它。正如第 7 章中讨论的，出于历史原因，我们通常将“**print**，打印”用作“**display**，显示”。

在 Unix 人士中，不管是否涉及到技术，都通常将“**grep**”当作动词使用。因此，您可能听到有人说“I lost your address, so I had to **grep** all my files to find your phone number(我丢了您的地址，所以我 **grep** 所有文件寻找您的电话号码)”或者“I **grepped** my living room twice, but I can't find the book you lent me(我在起居室 **grep** 了两遍，但是找不到您借给我的书)”。

作为一个有用的工具，**grep** 搜索单独的文件非常迅速，而它实现自身价值的地方就是管道线。这是因为 **grep** 可以从大量的原始数据中快速地归约出少量的有用信息。这是一个非常重要的功能，从而使 **grep** 成为 Unix 工具箱中一个十分重要的程序。询问任意一个有经验的 Unix 人士，都会发现他离不开 **grep**。理解这个神奇程序的强大功能需要花一段时间，但是我们可以从一些简单的例子入手。

当与其他人共享多用户系统时，可以使用 **w** 程序(参见第 8 章)显示所有用户以及他们正在做什么的相关信息。下面是一些样本输出：

```
8:44pm up 9 days, 7:02, 5 users, load: 0.11, 0.02, 0.00
User      tty      login@  idle   JCPU   PCPU   what
tammy     tty0     Wed10am 4days 42:41  37:56   -bash
harley    tty1     5:47pm      15:11      w
linda     tty3     5:41pm   10    2:16    13    -tcsh
casey     tty4     4:45pm      1:40    0:36   vi dogstuff
weedly    tty5     9:22am 1:40    20      1    gcc catprog.c
```

假设您希望显示在下午或者晚上登录系统的所有用户，那么您可以在输出中搜索包含模式 “pm” 的所有行。也就是说，使用下述管道线：

```
w -h | grep pm
```

注意 **w** 程序使用了 **-h** 选项。这将抑制掉标题，也就是不显示前两行。使用上面的数据，上一命令的输出为：

```
harley    tty1     5:47pm      15:11      w
linda     tty3     5:41pm   10    2:16    13    -tcsh
casey     tty4     4:45pm      1:40    0:36   vi dogstuff
```

假定我们只希望显示下午及晚上登录系统的用户标识。所需做的就是将 **grep** 的输出管道传送给 **cut**(参见第 17 章)，抽取数据的前 8 列：

```
w -h | grep pm | cut -c1-8
```

输出为：

```
harley
linda
casey
```

那么如何对输出排序呢？只需将输出管道传送给 **sort**(本章后面讨论)即可：

```
w -h | grep pm | cut -c1-8 | sort
```

输出为：

```
casey
harley
linda
```

19.2 最重要的 grep 选项

grep 程序有许多选项，但是我们只讨论最重要的选项。首先，我们讨论**-c**(count, 统计)选项，该选项显示所抽取行的数量，而不是所抽取的行本身。下面举例说明。

正如我们将在第 23 章中讨论的，Unix 文件系统使用目录，这与 Windows 和 Macintosh 中的文件夹相似(但并不完全相同)。目录中既可以包含普通文件，也可以包含其他目录(称为子目录)。例如，一个目录可能包含 20 个文件和 3 个子目录。

正如第 24 章所示，使用 **ls** 命令可以显示特定目录中包含的文件和子目录的名称。例如，下面的命令显示目录 **/etc** 的内容(阅读了第 23 章内容之后就会理解 **/etc** 的含义)。

```
ls /etc
```

如果在自己的系统上运行这个命令，就会发现 **/etc** 中包含许多条目。为了查看哪些条目是子目录，可以使用 **-F** 选项：

```
ls -F /etc
```

当使用这个选项时，**ls** 将在所有子目录的后面追加一个 **/**(反斜线)字符。例如，假设在输出中看到：

```
motd
rc.d/
```

那么这意味着 **motd** 是一个普通文件，而 **rc.d** 是一个子目录。

假定您希望统计 **/etc** 目录中所有子目录的数量。那么您需要做的就是将 **ls -F*** 的输出管道传送给 **grep -c**，并统计反斜线的数量：

```
ls -F /etc | grep -c "/"
```

在我的系统上，输出为：

```
92
```

顺便说一下，如果您希望统计目录中条目的总数，则只需将 **ls** 的输出管道传送给 **wc -l**(参见第 18 章)，例如：

```
ls /etc | wc -l
```

在我的系统上，**/etc** 目录中共有 242 个条目。

下一个选项是 **-i**，该选项告诉 **grep** 在进行比较时忽略大写字母和小写字母之间的区别。例如，假设 **food-costs** 文件中包含下述 5 行内容：

```
pizza $25.00
```

* 默认情况下，**ls** 在每行上列举多个文件名，从而使输出更加紧凑。但是，当将输出管道传送给另一个程序时，**ls** 将每个文件名显示在单独的行上。如果您希望模拟这一点，可以使用 **-l**(数字 1)选项，例如：

```
ls -l /etc
```

```
tuna $3.50
Pizza $23.50
PIZZA $21.00
vegetables $18.30
```

下述命令查找包含“pizza”的所有行。注意，根据 **grep** 的语法，模式应该位于文件名前面：

```
grep pizza food-costs
```

输出只有一行：

```
pizza $25.00
```

为了忽略大小写字母之间的区别，使用 **-i** 选项：

```
grep -i pizza food-costs
```

这一次，输出共有 3 行：

```
pizza $25.00
Pizza $23.50
PIZZA $21.00
```

提示

-i(ignore, 忽略)选项告诉 **grep** 忽略大写字母和小写字母之间的区别。在本章后面，我们将讨论其他两个程序 **look** 和 **sort**，它们也拥有相似的选项。但是，这两个程序使用的是 **-f**(fold, 同等)而不是 **-i**，不要混淆了。

(单词“fold”是一个技术术语，表示大写字母和小写字母应该同等对待。我们稍后讨论它。)

继续我们的讨论，有时候，您可能希望知道所选取的行在数据流中的位置。为此可以使用 **-n** 选项。这将告诉 **grep** 在输出的每一行前面写一个相对行号(数据不必包含行号，在处理输入的过程中，**grep** 会自动地统计行号)。作为示例，考虑下面的命令，该命令对上面的文件 **food-costs** 同时使用了 **-i** 和 **-n** 选项：

```
grep -in pizza food-costs
```

输出为：

```
1:pizza $25.00
3:Pizza $23.50
4:PIZZA $21.00
```

当需要确定特定行在大文件中的准确位置时，**-n** 选项非常有用。例如，假设您希望修改所有包含某特定模式的行。一旦借助 **grep -n** 查找到这些行的位置，就可以使用文本编辑器直接跳到希望进行修改的位置上。

下一个选项是 **-l**(list filename, 列举文件名)，当希望在不止一个文件中搜索特定模式时，

该选项非常有用。当使用 **-l** 选项时，**grep** 不显示包含该模式的各行，而是将包含这种模式的文件的名称写出来。

例如，假设您拥有 3 个文件：**names**、**oldnames** 和 **newnames**。文件 **names** 中包含有“harley”，文件 **newnames** 中包含有“Harley”，而文件 **oldnames** 两者都不包含。为了查看哪些文件包含模式“Harley”，可以使用：

```
grep -l Harley names oldnames newnames
```

输出为：

```
newnames
```

现在加上 **-i** 选项，忽略大小写字母之间的区别：

```
grep -il Harley names oldnames newnames
```

现在输出为：

```
names
```

```
newnames
```

-L(大写字母“L”)选项正好与 **-l** 选项相反。该选项显示不包含匹配模式的文件。在我们的例子中，为了列举不包含模式“Harley”的文件，可以使用：

```
grep -L Harley names oldnames newnames
```

输出为：

```
names
```

```
oldnames
```

下一个选项是 **-w**，该选项指定只希望搜索完整的单词。例如，假设有一个文件 **memo**，该文件包含下述内容：

```
We must, of course, make sure that all the
data is now correct before we publish it.
I thought you would know this.
```

您希望显示包含单词“now”的所有行。如果输入：

```
grep now memo
```

那么您将看到：

```
data is now correct before we publish it.
I thought you would know this.
```

这是因为 **grep** 既选取了“now”，也选取了“know”。如果您输入：

```
grep -w now memo
```

那么您将看到希望的输出:

```
data is now correct before we publish it.
```

-v(reverse, 相反)选项选取不包含指定模式的所有行。这是一个特别有用的选项,今后您会大量地使用。作为示例,假设您是一名学生,您有一个文件 **homework** 记录自己的作业。该文件为每个作业包含一行信息。一旦您完成一个作业,就将这个作业标记上“DONE”。例如:

```
Math: problems 12-10 to 12-33, due Monday
Basket Weaving: make a 6-inch basket, DONE
Psychology: essay on Animal Existentialism, due end of term
Surfing: catch at least 10 waves, DONE
```

为了列举所有还没有完成的作业,可以输入:

```
grep -v DONE homework
```

输出为:

```
Math: problems 12-10 to 12-33, due Monday
Psychology: essay on Animal Existentialism, due end of term
```

如果希望查看还没有完成的作业数量,可以组合使用**-c**和**-v**选项:

```
grep -cv DONE homework
```

在这个例子中,输出为:

```
2
```

有时候,可能希望查找那些完全由搜索模式构成的行。为此,可以使用**-x**选项。例如,假设文件 **names** 中包含下列各行:

```
Harley
Harley Hahn
My friend is Harley.
My other friend is Linda.
Harley
```

如果希望查找包含“Harley”的所有行,可以使用:

```
grep Harley names
```

如果只希望查找“Harley”占用整行的那些行,可以使用**-x**选项:

```
grep -x Harley names
```

在这种情况下, **grep** 只选取第一行和最后一行。

为了搜索整个目录树(参见第 23 章),可以使用**-r(recursive, 递归)**选项。例如,假设您

希望在目录 **admin** 下的所有文件中搜索单词 “initialize” —— 包括所有子目录以及所有子目录中的所有文件，可以使用：

```
grep -r initialize admin
```

当在大型目录树上使用 **-r** 选项时，通常会看到错误消息，告诉您 **grep** 无法读取某些特定的文件，这或者是因为文件不存在，或者是因为您没有权限读取它们(我们将在第 25 章中讨论文件权限)。一般情况下，您将看到下述两个消息中的一个：

```
No such file or directory
Permission denied
```

如果不希望看到这样的消息，可以使用 **-s**(suppress, 抑制)选项。例如，假设您以超级用户的身份登录，并且希望在系统上所有的文件中搜索单词 “shutdown now”。

正如我们将在第 23 章中讨论的，记号/指的是整个文件系统的根(主)目录。因此，如果从/目录开始，并且使用 **-r**(recursive, 递归)选项，那么 **grep** 将搜索整个文件系统。该命令如下所示：

```
grep -rs / 'shutdown now'
```

注意这里引用了搜索模式，因为它包含一个空格(引用在第 13 章解释过)。

19.3 grep 的变体：fgrep、egrep

以前(20 世纪 70 年代和 80 年代)，人们通常使用其他两个版本的 **grep**：**fgrep** 和 **egrep**。

fgrep 程序是一种快速版本的 **grep**，它只搜索“固定字符”的字符串(因此命名为 **fgrep**)。这意味着 **fgrep** 不允许使用正则表达式匹配模式。当计算机比较慢且内存有限时，只要您不需要正则表达式，那么 **fgrep** 就比 **grep** 更加高效。现在，计算机已经很快，并且拥有大量的内存，所以不再需要使用 **fgrep** 了。这里提及它主要是为了回顾历史。

egrep 程序是 **grep** 程序的扩展版本(因此命名为 **egrep**)。原始的 **grep** 只允许“基本的正则表达式”。而后来开发的 **egrep** 支持功能更强大的“扩展正则表达式”。我们将在第 20 章中讨论它们之间的区别。现在，您只需知道扩展正则表达式更好，在允许进行选择时，应该总是选择使用扩展正则表达式。

现代的 Unix 系统通过命令 **egrep** 或者 **grep -E** 允许使用扩展正则表达式。但是，大多数有经验的用户宁愿键入 **grep**。解决方法就是创建一个别名(参见第 13 章)，将 **grep** 改变成为 **egrep** 或者 **grep -E**。对于 Bourne shell 家族来说，可以使用下述命令中的一个：

```
alias grep='egrep'
alias grep='grep -E'
```

对于 C-Shell 家族来说，可以使用下述命令中的一个：

```
alias grep 'egrep'
```

```
alias grep 'grep -E'
```

一旦定义了这样一个别名，就可以在键入 **grep** 时获得扩展正则表达式的全部功能。为了使这样一个改变永久化，所需做的全部工作就是将合适的 **alias** 命令放到环境文件中(参见第 14 章)。实际上，这是一个非常有用的别名，因此建议您立即添加到自己的环境文件中。当阅读第 20 章时，我将假定您使用的就是扩展正则表达式。

注意：如果使用的是 Solaris(来自 Sun 公司)，那么您希望的 **egrep** 版本位于一个特殊的目录 `/usr/xpg4/bin/`^{*} 中，这意味着必须使用不同的别名。下面的例子只适用于 Solaris。第一个例子针对的是 Bourne shell 家族，第二个例子针对的是 C-Shell 家族：

```
alias grep='/usr/xpg4/bin/egrep'
alias grep '/usr/xpg4/bin/egrep'
```

19.4 选取以特定模式开头的行：look

相关过滤器：**grep**

look 程序搜索以字母顺序排列的数据，并查找所有以特定模式开头的行。

look 程序的使用方式有两种，即使用一个或多个文件中的有序数据，或者让 **look** 搜索一个字典文件(19.6 节中解释)。

当使用 **look** 搜索一个或多个文件时，语法为：

```
look [-df] pattern file...
```

其中 *pattern* 是搜索的模式，*file* 是文件的名称。

下面举例说明。您是一个学校的学生，每学期，所有的学生都要对教授进行评价。这个学期您负责该项目。您有一个大文件 **evaluations**，该文件中包含有对 100 多位教授的评价的汇总信息。数据按字母顺序排列。文件中的每行包含一个等级(A、B、C、D 或 F)，后面是两个空格，再后面就是教授的姓名。例如：

```
A William Wisenheimer
C Peter Pedant
F Norman Knowitall
```

您的任务就是创建 5 个列表，然后提交到网站上。这些列表应该包含获得 A 级、B 级等评价的教授的姓名。因为数据按字母顺序排列，所以可以使用 **look** 选取文件中所有以 A 开头的行，创建第一个列表(即评价为 A 级的教授)：

```
look A evaluations
```

尽管这条命令可以完成这项任务，但是我们依然可以改进它。正如前面所述，数据文件中的每行都以一个单独的等级字母开头，后面是两个空格。一旦拥有了希望的姓名列表，

^{*} 名称 **xpg4** 代表“X/Open Portability Guide, Issue 4”，它是一个定义 Unix 系统行为的旧标准(1992 年)。这个目录中的程序根据 XPG4 标准进行了修改。

就可以使用 **colrm**(参见第 16 章)移除每行的前 3 个字符。下述例子针对每个等级完成相应的任务，即从数据文件中选取合适的行，使用 **colrm** 移除每行的前 3 个字符，然后将输出重定向到文件：

```
look A evaluations | colrm 1 3 > a-professors
look B evaluations | colrm 1 3 > b-professors
look C evaluations | colrm 1 3 > c-professors
look D evaluations | colrm 1 3 > d-professors
look F evaluations | colrm 1 3 > f-professors
```

与本章中讨论的其他程序不同，**look** 不能从标准输入中读取数据，它必须从一个或多个文件中获取输入。这意味着，严格地讲，**look** 并不是一个过滤器。

有此限制的原因在于，对于标准输入来说，程序只能每次读取一行。但是，**look** 程序使用了一种称为“二分查找法”的搜索方法，要求同时访问所有的数据。基于这一原因，尽管可以在管道线的开头使用该程序，但是不能在管道线中间使用它。

当准备多步完成时，最好的策略就是准备好数据，将数据保存在文件中，然后再使用 **look** 搜索文件。例如，假设您有 4 个文件 **frosh**、**soph**、**junior** 和 **senior**，这 4 个文件包含上面描述的原始、无序的评价数据。在使用 **look** 搜索数据之前，必须组合并排序这 4 个文件的内容，然后将输出保存在一个新文件中，例如：

```
sort frosh soph junior senior > evaluations
look A evaluations
```

我们将在本章后面讨论 **sort** 程序。那时候，您将学习两个与 **look** 相关的特别选项。**-d**(dictionary, 字典)选项告诉 **sort** 只考虑字母和数字。当希望 **look** 忽略标点符号和其他特殊字符时，可以使用 **-d** 选项。**-f**(fold, 同等)选项告诉 **sort** 忽略大写字母和小写字母之间的区别。例如，当使用 **-f** 时，**sort** 将认为“Harley”和“harley”是相同的。

如果使用这两种 **sort** 选项准备数据，那么必须对 **look** 使用相同的选项，从而使 **look** 知道期望什么类型的数据。例如：

```
sort -df frosh soph junior senior > evaluations
look -df A evaluations
```

19.5 使用 look 和 grep 的时机

look 和 **grep** 都基于指定的模式从文本文件中选取行。基于这一原因，人们要问，什么时候使用 **look**，什么时候使用 **grep** 呢？

在许多场合中会出现相似的问题，因为 Unix 通常提供不止一种方法来解决一个问题。基于这一原因，明智地分析选项，从而能够挑选最好的工具来完成任务变得十分重要。作为示例，下面我们比较一下 **look** 和 **grep**。

look 程序在 3 个重要方面有限制。第一个方面，它要求有序输入；第二个方面，它只能从文件中读取数据，不能从标准输入中读取数据；第三个方面它只能在行的开头搜索指

定的模式。但是，在这些限制之外，**look** 有两大优点：易于使用并且速度非常快。

相对来说，**grep** 程序比较灵活：它不要求有序输入；它既能从文件中读取数据，也能从标准输入中读取数据(这意味着可以在管道线的中间使用它)；它还可以在任意位置搜索指定的模式，而不仅仅限于行的开头。

此外，**grep** 允许使用“正则表达式”，从而允许指定通用的模式，而不仅仅限于简单的字符串。例如，使用 **grep** 可以搜索“字符串 **har**，后面跟一个或多个字符，再后面跟字符串 **ley**，最后还有 0 个或多个数字”(正则表达式功能非常强大，我们将在第 20 章中详细讨论它)。

通过使用正则表达式，**grep** 能完成 **look** 可以完成的任何事情。但是 **grep** 比较慢，而且语法比较难使用。

因此下面就是我的建议：无论何时，当需要从文件中选取行时，先想一想 **look** 能不能完成任务。如果能，则使用 **look**，因为 **look** 既快又简单。如果 **look** 不能完成任务(大多数情况下是这样)，则使用 **grep**。通常，应该总是使用尽可能简单的解决方法解决问题。

但是如何考虑速度呢？前面提到过，**look** 比 **grep** 要快。这重要吗？在 Unix 初期，速度是重要的考虑事项，因为 Unix 系统由许多用户共享，而且计算机相对较慢。当从一个非常大的文件中选取文本行时，实际上能够注意到 **look** 和 **grep** 之间的速度差别。

但是，现在几乎所有的 Unix 系统所运行的计算机实际上都相当快。因此，Unix 执行一条单独命令的速度就无关紧要了，至少对于本章中的命令来说是这样的。例如，本章中的所有例子运行得都非常快，看上去都是瞬间完成的。更具体地讲，如果比较 **look** 命令和相应的 **grep** 命令，那么没有什么方法可以注意到它们之间速度上的差别。

因此我的建议就是基于简单性和易用性选择工具，而不必在意速度或效率上的微小差别。当编写程序(包括 shell 脚本)时，这尤为重要。如果程序或者脚本太慢，通常可以找到一两个影响速度的瓶颈，从而提高它们的速度。但是，如果程序的复杂或者难以使用的程度超出必要，那么从长远来看，它将浪费您的时间，相对于计算机时间来说，这样的代价更高。

提示

无论何时，当选择工具时，都应该使用能够完成任务的最简单工具。

19.6 查找以特定模式开头的所有单词：look

前面提到过可以使用 **look** 搜索字典文件。当希望查找所有以特定模式开头(例如，所有以字母“**simult**”开头的单词)的单词时就需要这么做。当以这种方式使用 **look** 时，**look** 程序的语法比较简单：

```
look pattern
```

其中 *pattern* 是搜索的模式。

“字典文件(dictionary file)”并不是一个实际字典。它是一个长的、综合的单词列表，

字典文件从 Unix 初期就一直存在(当然，该列表在一直不停地更新)。字典文件中的单词都是按字母顺序排列的，每行一个单词，因此可以方便地使用 **look** 进行搜索。

最初创建字典文件的目的是使用 **spell** 程序，该程序提供了一种检查文档拼写的简陋方法。**spell** 程序的任务就是显示文档中有但是字典文件中没有的单词清单。以前，**spell** 程序可以大量节省查找拼写错误的时间。

现在，已经有了更好的拼写检查工具，**spell** 程序就很少使用了。实际上，在大多数 Linux 或 Unix 系统中甚至都找不到它了。作为替代，人们或者使用字处理程序中的拼写检查特性，或者使用一个交互式程序 **aspell**(它是一个 GNU 实用工具)处理文本文件。如果希望试一试 **aspell**，可以使用：

```
aspell -c file
```

其中 *file* 是包含纯文本的文件的名称，**-c** 选项表明希望检查文件中单词的拼写。

尽管 **spell** 已经不再使用了，但是字典文件依然存在，您可以用多种方式使用它。特别地，您可以使用 **look** 程序查找所有以特定模式开头的单词。当遇到拼写问题时，使用该程序比较方便。例如，假设您希望键入单词“simultaneous”，但是不确定它的拼写，所以可以输入：

```
look simult
```

运行该命令后将看到一个类似于下述内容的列表：

```
simultaneity  
simultaneous  
simultaneously  
simultaneousness  
simulty
```

现在从中挑选出正确的单词就比较简单了，如果您希望，还可以将它从一个窗口复制粘贴到另一个窗口(有关如何复制和粘贴的内容请参见第 6 章)。

我们将在第 20 章中再次讨论字典文件，那时候，我们将示范在系统的什么地方查找实际文件，以及如何使用它解决单词难题。

(顺便说一下，“simulty”是指私人怨恨或争吵。)

提示

当使用 vi 文本编辑器(参见第 22 章)时，通过使用 **:r!** 可以发送一个快速的 **look** 命令，并显示一系列单词。例如：

```
:r !look simult
```

该命令将把所有以“simult”开头的单词插入到编辑缓冲区中。接下来就可以选择希望的单词，删除其他单词。

19.7 排序数据: sort

相关过滤器: **tsort**、**uniq**

sort 程序可以执行两个相关的任务: 排序数据以及查看数据是否已经有序。下面先从基本知识入手。排序数据的 **sort** 程序语法为:

```
sort [-dfnru] [-o outfile] [infile...]
```

其中 *outfile* 是存放输出的文件的名称, *infile* 是包含输入的文件名称。

sort 程序拥有极大的灵活性。它既可以比较整行, 也可以从每行中选取部分(字段)进行比较。使用 **sort** 的最简单的方法就是排序一个单独的文件、比较整行并在屏幕上显示结果。作为示例, 假设您有一个文件 **names**, 该文件中包含下述 4 行内容:

```
Barbara
Al
Dave
Charles
```

为了排序该数据并显示结果, 可以输入:

```
sort names
```

您将看到:

```
Al
Barbara
Charles
Dave
```

为了将有序数据保存到文件 **masterfile** 中, 可以重定向标准输出:

```
sort names > masterfile
```

上一个例子将有序数据保存在一个新文件中。但是, 许多时候希望将数据保存在相同的文件中。也就是说, 希望用排序后的相同数据替换原先的文件。然而, 不能使用一条命令将输出重定向到输入文件:

```
sort names > names
```

第 15 章中解释过当重定向标准输出时, **shell** 在运行命令之前建立输出文件。在这个例子中, 因为 **names** 是输出文件, 所以 **shell** 将在运行 **sort** 命令之前清空它。因此, 到 **sort** 程序准备读取输入时, **names** 已经是空的了。因此, 输入该命令将默默地清除输入文件的内容(除非已经设置了 **shell** 变量 **noclobber**)。

基于这一原因, **sort** 提供了一个特殊的选项, 允许将输出保存到任何希望的文件中。使用 **-o(output, 输出)** 选项, 后面跟着输出文件的名称, 如果输出文件与一个输入文件相同, 那么 **sort** 程序将为您保护数据。因此, 为了排序文件, 并将输出保存在同一个文件, 可以

使用类似于下面的命令：

```
sort -o names names
```

在这个例子中，**names** 中的原始数据将被保持，直到 **sort** 程序完成，然后输出才被写入到文件中。

为了排序多个文件中的数据，只需指定多个输入文件名。例如，为了排序文件 **oldnames**、**names** 和 **extranames** 的全部内容，并将输出保存到文件 **masterfile** 中，可以使用：

```
sort oldnames names extranames > masterfile
```

为了排序相同的文件，同时将输出保存在 **names**(输入文件之一)中，可以使用：

```
sort -o names oldnames names extranames
```

sort 程序通常用作管道线的一部分，用来处理由另一个程序生成的数据。下述例子组合两个文件，抽取那些包含字符串“Harley”的行，并进行排序，然后将输出发送给 **less** 进行显示：

```
cat newnames oldnames | grep Harley | sort | less
```

默认情况下，在排序数据时，**sort** 程序查看整行。但是，如果希望，也可以告诉 **sort** 只检查一个或多个字段，也就是行的一部分(我们在第 17 章中讨论 **cut** 程序时讨论了字段的概念)。这些选项允许在 **sort** 中使用字段，提供了极大的控制能力。但是，它们都非常复杂，因此这里将不深入讨论。如果希望对字段进行排序，那么您可以在 Info 文件(**info sort**)中查看详细内容。如果您的系统没有 Info 文件(参见第 9 章)，则可以在说明书页(**man sort**)中查看详细内容。

19.8 控制数据排序的顺序：sort -dfn

sort 程序有几个选项，可以用来控制 **sort** 程序的工作方式。

-d(dictionary, 字典)只查看字母、数字和空白符(空格和制表符)。当数据中包含可能妨碍排序过程的字符(例如标点符号)时，可以使用这个选项。

-f(fold, 等同)选项同等对待大写字母和小写字母。当希望忽略大写字母和小写字母之间的区别时，可以使用这个选项。例如，当使用 **-f** 选项时，**sort** 认为单词 **harley** 和 **Harley** 与 **HARLEY** 相同(术语“fold”在下面解释)。

-n(numeric, 数字)选项识别行开头或者字段开头的数字，并按数字进行排序。这样的数字可以包含前导 0、负号和小数点。使用这个选项可以告诉 **sort** 您使用的是数值数据。例如，假设您希望对下列数据排序：

```
11
2
```

```
1
20
10
```

如果使用没有选项的 **sort**，那么输出为：

```
1
10
11
2
20
```

如果使用的是 **sort -n**，那么输出为：

```
1
2
10
11
20
```

-r(reverse, 反向)选项按反向顺序对数据排序。例如，如果使用 **sort -nr** 对上一个例子中的数据排序，那么输出为：

```
20
11
10
2
1
```

以我的经验来看，**-r** 选项的使用次数要比您想象的多。这是因为按字母或者按数字的反向顺序列举信息非常有用。

最后一个选项是**-u(unique, 唯一)**，该选项告诉 **sort** 查找相同的行，并将相同的行只留下一行。例如，假设使用 **sort -u** 对下列数据排序：

```
Barbara
Al
Barbara
Barbara
Dave
```

那么输出为：

```
Al
Barbara
Dave
```

提示

作为 **sort** 程序**-u** 选项的一个备选方案，可以使用 **uniq**(本章后面讨论)程序。**uniq** 程序比较简单，但是与 **sort** 不同，它不能处理指定的字段，而这有时候是必须的。

名称含义

fold

Unix 中有许多程序都拥有忽略大写字母和小写字母之间区别的选项。有时候，该选项为 **-i**，代表“ignore，忽略”，这样比较合理。但是，大多数时候，该选项是 **-f**，代表“fold，同等”。它是一个技术术语，表示将小写字母看成大写字母，反之亦然，并且不修改原始数据(以这种方式使用术语“fold”与 **fold** 程序没有任何关系，因此不要混淆了)。

术语“fold”通常用作形容词：“为了使 **sort** 程序不区分大小写，使用 fold 选项。”但是有时候，也会看到“fold”用作动词：“当使用 **-f** 选项时，**sort** 程序将把小写字母 **fd** (看成)大写字母。”

下面是一些有趣的事情：Unix 中最初版本的 **sort** 程序将大写字母看成小写字母。也就是说，当使用 **-f** 选项时，**sort** 程序将所有的字母都看成是小写字母。现代版本的 **sort** 程序将小写字母看成大写字母。也就是说，它们将所有的字母都看成是大写字母。这个区别有意义吗？答案是有时候有，当讨论排序序列(collating sequence)时您就会明白。

没有人知道术语“fold”的起源，因此您可以自由地发挥想象。

19.9 检查数据是否有序：sort -c

正如前面所述，**sort** 程序可以执行两个相关的任务：数据排序以及检查数据是否已经有序。在本节中，将讨论数据的检查。当以这种方式使用 **sort** 程序时，它的语法为：

```
sort -c[u] [file]
```

其中 *file* 是文件的名称。

-c(check, 检查)选项告诉 **sort** 不希望排序数据，只希望知道数据是否已经有序。例如，为了查看文件 **names** 中的数据是否是有序的，可以使用：

```
sort -c names
```

如果数据是有序的，那么 **sort** 将不显示任何东西(没有消息就是好消息)。如果数据还不是有序的，那么 **sort** 将显示一个消息，例如：

```
sort: names:5: disorder: Polly Ester
```

在这个例子中，该消息意味着 **names** 中的数据不是有序的(也就是说，它们是“无序的”)，并且还说明从第 5 行开始，即从数据 **Polly Ester** 开始无序。

此外，还可以在管道线中使用 **sort -c** 检查由另一个程序写入到标准输出中的数据。例如，假设您有一个程序 **poetry-generator**，该程序生成大量的输出。假定输出是有序的，但是您怀疑程序可能会出现错误，因此使用 **sort -c** 检查该程序的输出：

```
poetry-generator | sort -c
```

如果组合使用 **-c** 选项和 **-u**(unique, 唯一)选项，那么 **sort** 将同时以两种方法检查数据。

除了查看无序数据，它还同时查看连续的相同行。当希望确保(1)数据是有序的，(2)所有行都是唯一的时，可以使用 **-cu** 选项。例如，文件 **friends** 中包含下述数据：

```
Al Packa
Max Out
Patty Cake
Patty Cake
Shirley U. Jest
```

输入：

```
sort -cu friends
```

尽管数据是有序的，但是 **sort** 检测到一个重复行：

```
sort: friends:4: disorder: Patty Cake
```

19.10 ASCII 码；排序序列

假定使用 **sort** 程序排序下述数据，那么输出将是什么呢？

```
zzz
ZZZ
bbb
BBB
aaa
AAA
```

在一些系统上，可能会得到：

```
AAA
BBB
ZZZ
aaa
bbb
zzz
```

而在另一些系统上，可能会得到：

```
AAA
aaa
BBB
bbb
ZZZ
zzz
```

怎么会是这种情况呢？在 Unix 初期，只有一种组织字符的方式。现在，情况已经不

是这样了。运行 **sort** 程序时得到的结果取决于您的系统中字符的组织方式。下面说明具体情形。

20 世纪 90 年代之前, Unix(以及大多数计算机系统)使用的字符编码都是 ASCII 码, 通常称为 ASCII。该名称代表 “American Standard Code for Information Interchange(美国信息交换标准码)”。

ASCII 码创建于 1967 年。它为每个字符指定的是 7 位的模式, 所以总共有 128 个字符。这些位模式的范围从 0000000(十进制 0)到 1111111(十进制 127)。基于这一原因, 128 个 ASCII 字符的编号从 0 到 127。

这 128 个字符构成的 ASCII 码包括 33 个“控制字符”和 95 个“可显示字符”。控制字符在第 7 章讨论过。可显示字符如下所示, 包括 52 个字母(26 个大写字母和 26 个小写字母)、10 个数字、32 个标点符号以及空格字符(列表中的第一个):

```
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

可显示字符的顺序按照上面列举的顺序排列。它们在 ASCII 码中的编码范围从字符 #32(空格)到字符 #126(波浪号)。记住, 编号从 0 开始, 因此空格实际上是第 33 个字符。作为参考, 附录 D 中包含有整个 ASCII 码的列表, 您可以看一看。

将制表符看成一个可显示的字符比较方便, 但严格地讲, 它实际上是一个控制字符。制表符是字符 #9, 位于其他可显示字符的前面。因此, 我提供下述定义: 96 个可显示字符是制表符、空格、标点符号、数字和字母。

为了方便起见, 大多数 Unix 系统都有一个参考页, 展示 ASCII 码, 以便您随时查看。然而, ASCII 参考页并不是标准化的, 因此显示 ASCII 参考页的方式取决于使用的系统, 详情请参见图 19-1。

Unix 类型	显示 ASCII 码参考页的命令
Linux	man ascii
FreeBSD	less /usr/share/misc/ascii
Solaris	less /usr/pub/ascii

图 19-1 显示 ASCII 码

本书附录 D 中提供有一个 ASCII 码的汇总信息。为了提供联机参考, 大多数 Unix 系统都拥有一个包含整个 ASCII 码的页面。传统上, 该页存储在目录 **/usr/pub/** 中的 **ascii** 文件中。最近几年, 一些系统重新组织了 Unix 文件系统, ASCII 参考文件被移到 **/usr/share/misc** 中。在其他系统上, 该文件已经转换成联机手册中的一个页面。因此, 显示 ASCII 参考页的方式取决于使用的系统。

在字符编码方案中, 字符组织的顺序称为**排序序列**。每当需要按顺序放置字符时——例如使用 **sort** 程序或者在正则表达式(第 20 章中讨论)中使用范围, 都会涉及到排序序列。

对于 ASCII 码而言, 排序序列只是字符出现在编码中的顺序, 如图 19-2 所示。更多的细节, 请参见附录 D。

编号	字符
0-31	控制字符(包括制表符)
32	空格字符
33-47	符号: !"#\$%&'()*+,-./
48-57	数字: 0 1 2 3 4 5 6 7 8 9
58-64	更多符号: ; < = > ? @
65-90	大写字母: A B C ... Z
91-96	更多符号: [\] ^ _ `
97-122	小写字母: a b c ... z
123-126	更多符号: { } ~
127	null 控制字符(del)

图 19-2 ASCII 码中的字符顺序

ASCII 码定义了 Unix 系统使用的 128 个基本字符。在 ASCII 码中，字符的编号从 0 至 127。该图中的表汇总了字符的顺序，这对使用诸如 **sort** 之类的程序来说非常重要。例如，当排序文本时，空格位于“%”(百分号)的前面，而“%”位于数字“3”的前面，“3”又位于字母“A”的前面，等等。更多的细节信息，请参见附录 D。

熟悉 ASCII 码的排序序列非常重要，因为许多 Unix 系统和编程语言都默认使用它。尽管不必记住整个 ASCII 码，但是需要记住 3 个基本原则：

- 空格在数字之前
- 数字在大写字母之前
- 大写字母在小写字母之前

下面举例说明。假定您的系统使用 ASCII 排序序列，您使用 **sort** 程序对下述数据进行排序(第 3 行以空格开头)：

```
hello
Hello
 hello
1hello
:hello
```

那么输出为：

```
hello
1hello
:hello
Hello
hello
```

提示

当需要 ASCII 码中的字符顺序时，只需记住以下顺序：空格、数字、大写字母和小写

字母。也就是记住“SNUL”。

19.11 区域设置和排序序列

在 Unix 初期，每个人都使用 ASCII 码，因此也没有什么问题。但是，ASCII 基于英语，而随着 Unix、Linux 以及 Internet 在世界范围内的扩展，有必要设计一种新系统，从而能够处理许多种语言以及大量不同的文化习俗。

20 世纪 90 年代，开发人员开发出了一种新系统，该系统基于“区域设置(locale)”的思想，它属于 POSIX 1003.2 标准(POSIX 在第 11 章和第 16 章中讨论过)。区域设置是一种技术规范，描述与特定文化的用户通信时应该使用的语言和习俗。该系统的意图就是用户根据自己的需要选择一个区域设置，然后用户运行的程序据此与他进行通信。例如，如果用户选择了美国英语区域设置，那么他的程序将以英语显示消息，以格式“月-日-年”写入日期，使用“\$”作为货币符号等。

在 Unix 中，区域设置由一组标识语言、日期格式、时间格式、货币符号和其他文化习俗的环境变量所定义。无论何时，当程序需要知道个人偏爱时，程序只需查看合适的环境变量。特别地，有一个环境变量 **LC_COLLATE** 指定使用哪一种排序序列(各个变量都有默认值，可以根据需要进行修改)。

为了显示系统中所有区域设置变量的当前值，包括 **LC_COLLATE**，可以使用 **locale** 命令：

```
locale
```

如果想知道自己的系统支持哪些区域设置，可以使用 **-a(all, 全部)** 选项显示它们：

```
locale -a
```

在美国，Unix 系统默认设置为两个区域设置中的一个。两个区域设置基本上是相同的，但是有不同的排序序列，这意味着当运行 **sort** 之类的程序时，结果会根据使用的区域设置不同而有所变化。

因为许多人没有意识到区域设置的存在，所以即便是有经验的程序员，当从一个 Unix 系统切换到另一个 Unix 系统时，也有可能感到迷惑，因为突然之间，熟悉的程序(如 **sort**)的工作“不正常”了。基于这一原因，下面讨论一下美国的两种区域设置。如果您没居住在美国，那么这些思想仍然适用，只是细节有所不同。

第一种美国区域设置基于 ASCII 码。这种区域设置有两个名称。它被称为是 **C** 区域设置(跟随 C 编程语言命名)或 **POSIX** 区域设置。两个名称可以任意使用。第二种美国区域设置基于美国英语，命名为 **en_US**，但是名称有许多变化。

C 区域设置是为兼容性设计的，即为了保留以前的程序(以及以前的程序员)所使用的

* 如果您记不住缩写词 SNUL，那么我示范一个许多聪明人使用的记忆技巧。您只需将需要记住的东西与日常生活中的东西联系起来。

例如，假设您是一名数学家，专攻微分计算，您碰巧在处理 4 阶微分方程式，该方程式满足与特殊非一致性点阵正交的 Laguerre-Hahn 多项式。那么为了记住 SNUL 的含义，只需将 SNUL 认为是“special non-uniform lattices(特殊非一致性点阵)”的缩写即可。

看看成为聪明人是多么的容易？

习俗。**en_US** 区域设置是为适应现代国际架构设计的，在该架构中美国英语只是众多不同语言中的一种。

正如前面所述，除了排序序列不同外，这两种区域设置是相同的。**C** 区域设置使用 ASCII 排序序列，在这种排序序列中，大写字母位于小写字母之前：**ABC...XYZabc...z**。该模式称为 **C 排序序列**，因为 C 编程语言使用它。

en_US 区域设置使用一种不同的排序序列，在这种排序序列中，小写字母和大写字母成对分组：**aAbBcCdD...zZ**。这种模式比较自然，因为它以字典顺序组织单词和字符。基于这一原因，将这一模式称为字典排序序列(dictionary collating sequence)。

直到 20 世纪 90 年代末，所有的 Unix 系统都使用 C 排序序列，该排序序列基于 ASCII 码，而且使用 **C/POSIX** 区域设置的系统仍然采用这一编码。但是，现在一些 Unix 系统，包括少数几个 Linux 分发版，在设计上都追求更多地体现国际化。因此，它们使用 **en_US** 区域设置和字典排序序列。

您有没有发现潜在的混乱？每当运行依赖于大写字母和小写字母顺序的程序时，程序的输出都会受排序序列的影响。因此，根据系统默认使用的区域设置的不同，可能得到不同的结果。例如，当使用 **sort** 程序，或者使用特定类型的正则表达式(称为“字符类”，参见第 20 章)时，可能会发生这种情况。

出于参考目的，图 19-3 示范了这两种排序序列。注意它们之间有明显的不同，不仅包括字母的顺序，而且还有标点符号的顺序。

C 区域设置: C 排序序列	
空格字符	.
符号	!"#\$%&'()*+,-./
数字	0123456789
更多符号	;<=>?@
大写字母	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
更多符号	[\]^_`
小写字母	a b c d e f g h i j k l m n o p q r s t u v w x y z
更多符号	{ } ~

en_US 区域设置: 字典排序序列	
符号	` ^ ~ < = >
空格字符	.
更多符号	_ - , ; : ! ? / . ' " () [] { } @ \$ % & ' # % +
数字	0 1 2 3 4 5 6 7 8 9
字母	a A b B c C d D e E f F g G h H i I j J k K l L m M n N o O p P q Q r R s S t T u U v V w W x X y Y z Z

图 19-3 C 区域设置和 en_US 区域设置的排序序列

在美国，Unix 系统和 Linux 系统使用下述两种区域设置中的一个：基于 ASCII 码的 **C/POSIX**，或者基于美国英语的 **en_US**。这两张表分别示范了每一种区域设置的排序序列。在 C 排序序列(由 C 区域设置使用)中，数字、小写字母和大写字母分别被符号分隔开。在字典排序序列(由 en_US 区域设置使用)中，所有的符号都位于开头，后面是数字和字母。

注意：在两张图中，我使用了点号(.)来表示空格字符。

作为区域设置如何产生不同结果的一个例子，下面我们考虑在对下述数据(在这些数据中，第3行以空格开头)进行排序时，会发生什么情况：

```
hello
Hello
  hello
lhello
:hello
```

对于 C 区域设置(C 排序序列)来说，输出为：

```
  hello
lhello
:hello
Hello
hello
```

对于 en_US 区域设置(字典排序序列)来说，输出为：

```
  hello
:hello
lhello
hello
Hello
```

那么应该使用哪一种区域设置呢？从我的经验来看，如果使用 en_US 区域设置，那么最终会遇到不可预测的问题，并且难以追查。例如，正如第 25 章所讨论的，可以使用 **rm**(remove, 移除)程序删除文件。假设您希望删除所有以大写字母开头的文件名，则所使用的传统 Unix 命令为：

```
rm [A-Z]*
```

如果使用的是 C 区域设置，那么这没有什么问题。但是，如果使用的是 en_US 区域设置，那么最终将删除所有以大写字母或小写字母开头的文件——除了字母 **a** 开头的文件(现在还不用考虑细节问题，第 20 章将详细解释)*。

建议将区域设置默认设置为 C 区域设置，因为它使用传统的 ASCII 排序序列。从长远来看，这比使用 en_US 区域设置和字典排序序列产生的问题更少。实际上，在阅读本书时，我假定您使用的是 C 区域设置。

那么如何指定区域设置呢？第一步就是确定系统默认使用哪一种排序序列。如果 C 区域设置已经是系统上的默认区域设置，那么没有什么问题；如果不是，则需要改变它。

一种确定默认排序序列的方式就是输入 **locale** 命令，然后检查 **LC_COLLATE** 环境变量

* 在许多场合中，C 区域设置要比 en_US 区域设置好。下面再举一个例子，假设您正在编写 C 或者 C++ 程序。在您的目录中，有一些代码文件的名称都是由小写字母构成的，例如 **program1.c**、**program2.cpp**、**data.h** 等。还有一些文件的名称是以大写字母开头的，例如 **Makefile**、**RCS**、**README** 等。当使用 **ls** 程序(参见第 24 章)列举目录中的内容时，所有的“大写字母”文件首先列举，从而将其他文件与代码文件分开列举。

量的值，看看它是 **C** 或 **POSIX**，还是某种形式的 **en_US**。

另一种确定默认排序序列的方式就是执行下面的简短测试。使用下述命令创建一个小文件 **data**：

```
cat > data
```

键入下述 3 行，然后按 **^D** 键结束命令：

```
AAA
[]
aaa
```

接下来排序文件的内容：

```
sort data
```

如果使用的是 **C/POSIX** 区域设置，那么输出将使用 **C(ASCII)** 排序序列排序：

```
AAA
[]
aaa
```

如果使用的是 **en_US** 区域设置，那么输出将使用字典排序序列排序：

```
[]
aaa
AAA
```

在继续之前，请看看图 19-3 中的排序序列，以确保理解了这些例子。

如果 Unix 系统默认使用 **C** 或 **POSIX** 区域设置，则不需要做任何事情(但是，请通读本节的剩余部分，可能有一天，您会在另一个系统上遇到这种问题)。

如果系统使用的是 **en_US** 区域设置，那么您需要将 **LC_COLLATE** 环境变量改变成 **C** 或者 **POSIX**。在 Bourne Shell 家族中，下面两条命令中的任何一条都可以完成该任务：

```
export LC_COLLATE=C
export LC_COLLATE=POSIX
```

对于 C-Shell 家族，则可以使用：

```
setenv LC_COLLATE C
setenv LC_COLLATE POSIX
```

为了永久改变这个环境变量，只需将这些命令中的一条放在登录文件中(环境变量在第 12 章中讨论，登录文件在第 14 章中讨论)。在本书的剩余部分，我们假定您使用的是 **C** 排序序列，因此，如果不是这样，那么您需要立即在登录文件中放入合适的命令。

提示

有时候，可能需要运行一个单独的程序，该程序需要使用与默认设置不同的排序序列。

为了这样做，在运行该程序时，需要使用一个子 shell 临时改变 `LC_COLLATE` 环境变量的值(有关子 shell 的讨论请参见第 15 章)。

例如，假设您使用的是 C 区域设置，而您希望使用 `en_US`(字典)排序序列运行 `sort` 程序。那么您可以使用：

```
(export LC_COLLATE=en_US; sort data)
```

当以这种方式运行程序时，对 `LC_COLLATE` 变量的改变是临时的，因为它只存在于子 shell 中。

19.12 查找重复行：uniq

相关过滤器：`sort`

Unix 中有许多专门的过滤器用来处理有序数据。其中最有用的过滤器就是 `uniq`，`uniq` 程序一行一行地检查数据，查找连续重复的行。

`uniq` 程序可以执行 4 项不同的任务：

- 消除重复行
- 选取重复行
- 选取唯一行
- 统计重复行的数量

`uniq` 程序的语法为：

```
uniq [-cdu] [infile [outfile]]
```

其中 *infile* 是输入文件的名称，*outfile* 是输出文件的名称。

首先从一个简单的例子开始。文件 `data` 包含下面各行：

```
Al
Al
Barbara
Barbara
Charles
```

(记住，因为 `uniq` 的输入必须是有序的，所以重复行是连续的。)

假设您希望生成包含该文件的所有行，但没有重复行的输出，所使用的命令为：

```
uniq data
```

输出相当直观：

```
Al
Barbara
Charles
```

如果希望将输出保存到另一个文件中，比如说 **processed-data**，则可以将该文件的名称指定为命令的一部分：

```
uniq data processed-data
```

要只查看重复行，可以使用 **-d** 选项：

```
uniq -d data
```

使用我们的样本文件，输出为：

```
Al
Barbara
```

为了只查看唯一(非重复)行，可以使用 **-u** 选项：

```
uniq -u data
```

在我们的例子中，只有一个这样的行：

```
Charles
```

问题：如果同时使用 **-d** 和 **-u** 选项，您认为会发生什么情况呢？试试看。
为了统计每行重复出现的次数，可以使用 **-c** 选项：

```
uniq -c data
```

对于我们的例子来说，输出为：

```
2 Al
2 Barbara
1 Charles
```

到目前为止，我们的例子都是简单的。当在管道线中使用 **uniq** 程序时，**uniq** 程序的真正威力才会体现出来。例如，经常需要组合并排序几个文件，然后将输出管道传送给 **uniq**，如下述两个例子所示：

```
sort file1 file2 file3 | uniq
cat file1 file2 file3 | sort | uniq
```

提示

如果不带选项地使用 **uniq** 程序，那么还有一种替代方法，即使用 **sort -u**。例如，下述 3 条命令实现相同的效果：

```
sort -u file1 file2 file3
sort file1 file2 file3 | uniq
cat file1 file2 file3 | sort | uniq
```

(参见本章前面对 **sort -u** 的讨论。)

下面是一个真实生活的例子，可以显示出此类构造的功能有多强大。

Ashley 是南加利福尼亚州一所大型学校的学生。在即将到来的寒假中，她的堂姐 Jessica 将从东海岸过来拜访她，她准备去一所优秀的小型文科学校。Jessica 想和男孩子约会，但是她特别挑剔：她只喜欢那些非常聪明并且还是运动员的男孩子。

碰巧 Ashley 在女生联谊会的道德委员会中任职，因此她能够访问学生/成绩数据库(不要问我她是如何访问的)。通过使用自己的特殊身份，Ashley 登录到系统中，创建了两个文件。第一个文件是 **math237**，该文件中包含所有选修了数学 237 课程(高级微积分学)的男学生的姓名。第二个文件是 **pe35**，该文件中包含所有选修体育 35 课程(冲浪鉴赏)的男学生的姓名。

Ashley 的想法就是通过查找所有选修这两门课程的男孩，生成一串潜在的约会对象。因为这两个文件对于手工比较来说非常大，所以 Ashley(既漂亮又聪明的一个女孩)决定使用 Unix。具体而言，就是使用带选项 **-d** 的 **uniq** 程序，并将输出保存在文件 **possible-guys** 中：

```
sort math237 pe35 | uniq -d > possible-guys
```

然后 Ashley 将这个列表通过电子邮件发送给 Jessica，使得 Jessica 可以在启程之前在 Myspace 上核对这些男孩。

提示

uniq 的输入必须是有序的。如果不是，那么 **uniq** 就不能够检测重复行，从而无法生成期望的结果，而且在这种情况下，没有提醒出现错误的警告信息。^{*}

19.13 合并两个文件中的有序数据：join

相关过滤器：**colrm**、**cut**、**paste**

在所有针对有序数据专门设计的 Unix 过滤器中，最有趣的过滤器就是 **join** 了，它基于特定字段的值将两个有序文件组合在一起。**join** 程序的语法为：

```
join [-i] [-a1|-v1] [-a2|-v2] [-1 field1] [-2 field2] file1 file2
```

其中，*field1* 和 *field2* 是引用特定字段的数字，*file1* 和 *file2* 是包含有序数据的文件的名称。

在详细介绍之前，先示范一个例子。假设您有两个有序文件，这两个文件都包含有许多人的信息，其中每个人都有唯一的标识号。在第一个文件 **names** 中，每行包含一个 ID 号以及这个人的姓氏和名字。

```
111 Hugh Mungus
222 Stew Pendous
```

^{*} 再一次说明了“有序”的事情才是好事情，即便对 Ashley 和 Jessica 来说。

```
333 Mick Stup
444 Melon Collie
```

在第二个文件 **phone** 中，每行包含一个 ID 号以及一个电话号码：

```
111 101-555-1111
222 202-555-2222
333 303-555-3333
444 404-555-4444
```

join 程序允许基于两个文件中共同的值组合文件，在这个例子中，这个共同的值就是 ID 号：

```
join names phone
```

输出为：

```
111 Hugh Mungus 101-555-1111
222 Stew Pendous 202-555-2222
333 Mick Stup 303-555-3333
444 Melon Collie 404-555-4444
```

当 **join** 读取输入时，它忽略前导空白符，也就是行头的空格和制表符。例如，**join** 程序认为下述两行是相同的：

```
111 Hugh Mungus 101-555-1111
  111 Hugh Mungus 101-555-1111
```

在详细讨论 **join** 程序之前，首先复习几个术语。在第 17 章中，我们讨论了字段和定界符。当拥有某个文件，它的每行都包含有一个数据记录时，行中的每个单独项是一个字段。在我们的例子中，文件 **names** 中的每行包含三个字段：ID 号、名字和姓氏。文件 **phone** 中包含两个字段：ID 号和电话号码。

在每行中，分隔各个字段的字符称之为定界符。在我们的例子中，定界符就是空格，通常以这种方式使用的还有制表符和逗号。默认情况下，**join** 假定各个字段之间用空白符分隔，也就是说用一个或多个空格或制表符分隔。

当基于匹配的字段组合两组数据时，我们称之为**联接**(该名称来自数据库理论)。用来匹配的具体字段称为**联接字段**(join field)。默认情况下，**join** 假定联接字段是每个文件的第一个字段，但是，正如您稍后所看到的，这可以改变。

为了创建一个联接，程序需要在两个文件中查找行对，也就是说联接字段拥有相同值的行。对于每个行对来说，**join** 生成一个包含 3 部分的输出：公共联接字段值、第一个文件中行的其余部分以及第二个文件中行的其余部分。

作为示例，考虑上面两个文件中的第一行。联接字段的值为 **111**。因此，输出的第一行由 **111**、一个空格、**Hugh**、一个空格、**Mungus**、一个空格和 **101-555-1111** 构成(默认情况下，**join** 使用一个空格分隔输出中的字段)。

在上面的例子中，第一个文件中的每一行都与第二个文件中的某一行匹配。但是，现

实可能并不总是这种情况。例如，考虑下述文件。您在生成一个朋友们的生日及喜爱礼物的列表。第一个文件 **birthdays** 中包含有两个字段，名字和生日：

```
Al      May-10-1987
Barbara Feb-2-1992
Dave    Apr-8-1990
Frances Oct-15-1991
George  Jan-17-1992
```

第二个文件 **gifts** 中也包含有两个字段，名字和礼物：

```
Al      money
Barbara chocolate
Charles music
Dave    books
Edward  camera
```

在这个例子中，Al、Barbara 和 Dave 的生日信息和礼物信息都有了。但是 Frances 和 George 没有礼物信息，而 Edward 没有生日信息。对它们使用 **join** 程序，会发生什么情况？

```
join birthdays gifts
```

因为只有 3 行拥有匹配的联接字段(即 Al、Barbara 和 Dave 的行)，所以输出只有 3 行：

```
Al May-10-1987 money
Barbara Feb-2-1992 chocolate
Dave Apr-8-1990 books
```

但是，假定您希望查看所有人的生日信息，即便他们没有礼物信息，则可以使用 **-a(all, 全部)** 选项，后面跟一个 1：

```
join -a1 birthdays gifts
```

这将告诉 **join** 输出第 1 个文件中的所有名字，即使他们没有礼物信息：

```
Al May-10-1987 money
Barbara Feb-2-1992 chocolate
Dave Apr-8-1990 books
Frances Oct-15-1991
George Jan-17-1992
```

同样，如果希望了解所有人的礼物信息(第 2 个文件中)，即使他们没有生日信息，则可以使用 **-a2**：

```
join -a2 birthdays gifts
```

输出为：

```
Al May-10-1987 money
Barbara Feb-2-1992 chocolate
```

```
Charles music
Dave Apr-8-1990 books
Edward camera
```

为了列举两个文件中的所有名字，可以同时使用两个选项：

```
join -a1 -a2 birthdays gifts
```

输出为：

```
Al May-10-1987 money
Barbara Feb-2-1992 chocolate
Charles music
Dave Apr-8-1990 books
Edward camera
Frances Oct-15-1991
George Jan-17-1992
```

如第一个例子一样，当以常规方式使用 **join**(没有 **-a** 选项)时，所获得的结果称为内联接(inner join, 该术语取自数据库理论)。对于内联接来说，输出只包含联接字段匹配的行。

当使用 **-a1** 或 **-a2** 选项时，输出还包含联接字段不匹配的行。我们称这种输出为外联接(outer join)。

这里不准备深入展开讨论，虽说数据库理论非常有趣，但是它已经超出了本书的范围。我只希望您记住，如果使用所谓的“关系数据库”，那么内联接和外联接之间的区别非常重要。

如果只希望查看那些不匹配的行，则可以使用 **-v1** 或者 **-v2**(reverse, 相反)选项。当使用 **-v1** 选项时，**join** 程序只输出第 1 个文件中不匹配的行，而忽略匹配的行。例如：

```
join -v1 birthdays gifts
```

输出为：

```
Frances Oct-15-1991
George Jan-17-1992
```

当使用 **-v2** 选项时，结果是第 2 个文件中不匹配的行：

```
join -v2 birthdays gifts
```

输出为：

```
Charles music
Edward camera
```

当然，也可以同时使用两个选项，获得两个文件中不匹配的行：

```
join -v1 -v2 birthdays gifts
```

现在输出为：


```
Charles music
Edward camera
Frances Oct-15-1991
George Jan-17-1992
```

因为 **join** 依赖于数据是否是有序的，所以还有几个选项帮助控制排序要求。首先，可以使用 **-i(ignore, 忽略)** 选项告诉 **join** 忽略大写字母和小写字母之间的区别。例如，当使用这个选项时，程序将认为 **CHARLES** 与 **Charles** 相同。

提示

通常使用 **sort** 程序为 **join** 准备数据。记住：对于 **sort** 来说，通过使用 **-f(fold, 同等)** 选项可以忽略大写字母和小写字母之间的区别。对于 **join** 来说，可以使用 **-i(ignore, 忽略)** 选项(参见本章前面有关“fold”的讨论)。

正如前面所述，**join** 假定联接字段是每个文件的第一个字段。通过使用 **-1** 和 **-2** 选项可以指定使用不同的联接字段。

为了改变第 1 个文件的联接字段，可以使用 **-1** 选项，后面跟希望使用的字段的编号。例如，下述命令联接两个文件 **data** 和 **statistics**，所使用的联接字段是第 1 个文件的第 3 个字段和第 2 个文件的第 1 个字段(默认)：

```
join -1 3 data statistics
```

为了改变第 2 个文件的联接字段，可以使用 **-2** 选项。例如，下述命令使用第 1 个文件的第 3 个字段和第 2 个文件的第 4 个字段进行联接：

```
join -1 3 -2 4 data statistics
```

最后，我希望提醒您，因为 **join** 使用的是有序数据，所以获得的结果依赖于区域设置和排序序列，也就是环境变量 **LC_COLLATE** 的值(有关区域设置的信息请参见本章前面的讨论)。

提示

在使用 **join** 程序的过程中，最常见的错误就是没有对两个输入文件进行排序。如果一个或两个文件没有根据联接字段正确排序，则结果或者是没有输出，或者只是部分输出，而且没有错误消息警告出现了问题。

19.14 由偏序创建全序：tsort

相关过滤器：**sort**

考虑下述问题。您正在安排晚上的活动，但是有许多约束条件：

- 器皿必须在看电视之前清洗。
- 吃饭必须在清洗器皿之前。

- 购物必须在做晚饭之前。
- 购物必须在准备食物之前。
- 食物必须在做晚饭之前准备。
- 晚饭必须在吃饭之前做。
- 食物必须在看电视之前准备。

可以看出，这多少有点混乱。您需要的是一个主列表，指定什么活动应该在什么时候完成，而且这些约束必须都得到满足。

从数学术语上讲，每个约束都称为一个**偏序**(partial ordering)，因为它们只指定了一些(不是全部)活动的顺序。在我们的例子中，每个偏序指定两个活动的顺序。您能据此构建一个主列表吗？主列表是**全序**(total ordering)的，因为它指定全部活动的顺序。

tsort 程序的任务就是分析一组偏序，其中每个偏序都代表一个约束，并计算满足全部约束的全序。**tsort** 程序的语法很简单：

```
tsort [file]
```

其中 *file* 是文件的名称。

输入的每一行必须包含一对由空白符(空格或者制表符)分隔的字符串，每一对字符串都代表一个偏序。例如，假设文件 **activities** 中包含下述数据：

```
clean-dishes watch-TV
eat clean-dishes
shop cook
shop put-away-food
put-away-food cook
cook eat
put-away-food watch-TV
```

注意该文件中每行包含两个由空白符分隔(在这个例子中是一个空格)的字符串。每行表示一个与上述某个约束条件对应的偏序。例如，第一行是说必须在看电视之前清洗器皿，第二行是说必须在清洗器皿之前吃饭，等等。

tsort 程序将一组偏序转换成一个全序。使用命令：

```
tsort activities
```

输出为：

```
shop
put-away-food
cook
eat
clean-dishes
watch-TV
```

因此，该问题的解决方案是：

- (1) 购物。

- (2) 准备食物。
- (3) 做晚饭。
- (4) 吃饭。
- (5) 清洗器皿。
- (6) 看电视。

通常，任何偏序组都可以组合成一个全序，只要偏序组中没有循环。例如，考虑下述偏序：

```
study watch-TV
watch-TV study
```

这种情况没有全序，因为如果坚持在学习之前看电视的话，则不能在看电视前学习(尽管大多数人试图能够这样做)。如果将该数据发送给 **tsort** 程序，它将显示一个错误消息，说明输入中包含有循环。

名称含义

tsort

从数学上讲，可以使用所谓的“有向图”表示一组偏序。如果这组偏序中没有循环，则可以称它为“有向无环图(directed acyclic graph, DAG)”。例如，树(参见第9章)就是一个 DAG。

一旦拥有了 DAG，就可以基于元素在图中的相对位置(不是元素的值)对元素排序，从而由偏序创建全序。实际上，这就是 **tsort** 程序完成任务的方式(我们并不需要了解细节)。

在数学中，我们使用单词“topological(拓扑)”描述依赖于相对位置的属性。因此，**tsort** 代表“topological sort(拓扑排序)”。

19.15 在二进制文件中搜索字符串：strings

相关过滤器：grep

为了使用 **strings** 程序，需要理解文本文件和二进制文件之间的区别。首先考虑下面 3 个定义：

(1) 可显示字符有 96 个：制表符、空格、标点符号、数字和字母。任何一个可显示字符序列都称为一个字符串(character string)，或者非正式地简化为串(string)。例如，“Harley”就是一个长度为 6 的字符串(本章前面讨论了可显示字符)。

(2) 只包含可显示字符的文件(每行结束都有一个换行符)称为文本文件或者 ASCII 文件。大多数情况下，Unix 的过滤器都是为处理文本文件设计的。实际上，在本章中，所有的示例文件都是文本文件。

(3) 二进制文件是任何非文本文件的非空文件，也就是指任何至少包含少量非文本数据的文件。二进制文件的常见例子包括可执行程序、对象文件、图像、声音文件、视频文件、字处理程序文档、电子表格和数据库。

如果您是程序员，那么您有可能使用过可执行程序 and 对象文件(程序的“片段”)，它们都是二进制文件。如果查看可执行程序或者对象文件的内部，那么看到的大部分内容是编码的机器指令，它们看上去像是没有意义的无用信息。但是，大多数程序也包含一些可识别的字符串，例如错误消息、帮助信息等。

strings 程序就是为了让程序员显示嵌在可执行程序 and 对象文件中的字符串而创建的一个工具。例如，程序员习惯在每个程序中插入一个字符串，显示程序的版本。这将允许任何人使用 **strings** 程序从程序本身中抽取程序的版本信息。

现在，程序员和用户拥有获取此类信息的更好方法^{*}，因而 **strings** 程序已经不太经常使用。然而，依然可以使用它，或许只是为了好玩而看看任何类型二进制文件的“内部”。尽管极少有实际原因需要这样做，但是能够检查二进制文件的内部，寻找隐藏信息也是很酷的。**strings** 程序的语法为：

```
strings [-length] [file...]
```

其中 *length* 是要显示的字符串的最小长度，*file* 是文件的名称，通常是一个路径名。

作为示例，假设您希望查看 **sort** 程序的内部。首先，使用 **whereis** 程序查找包含该程序的文件的路径名，也就是其准确位置(我们将在第 24 章中讨论路径名和 **whereis**，因此现在在还不用考虑细节问题)。使用的命令为：

```
whereis sort
```

典型输出为：

```
sort: /usr/bin/sort /usr/share/man/man1/sort.1.gz
```

输出显示该程序的准确位置和说明书页。我们只对程序感兴趣，为了使用 **strings** 查看 **sort** 程序的内部，我们使用：

```
strings /usr/bin/sort
```

这样的命令通常生成许多输出。但是，有 3 件事情可以使输出更加便于管理。

首先，也是默认情况，**strings** 只抽取至少有 4 个字符长的字符串。该方式可以消除那些没有意义的短字符序列。即便是这样，您也有可能看到大量的无用字符串。但是，通过指定一个更长的最小长度可以消除更多无用字符串。要实现该目的，可以使用一个由连字符(-)和数字构成的选项。例如，为了指定查看至少包含 7 个字符的字符串，可以使用：

```
strings -7 /usr/bin/sort
```

接下来，可以对输出排序，移除重复行。要实现该目的，只需将输出管道传送给 **sort** -iu(本章前面讨论过)：

```
strings -7 /usr/bin/sort | sort -iu
```

^{*} 正如第 10 章中讨论的，大多数 GNU 实用工具(Linux 和 FreeBSD 中使用)支持 **--version** 选项显示程序的版本信息。

最后，如果输出很多，以至于在阅读之前就滚动出屏幕范围，则可以使用 **less**(参见第 21 章)命令每次一屏地显示输出：

```
strings -7 /usr/bin/sort | sort -iu | less
```

如果查看程序内部隐藏信息的想法对您有吸引力，那么下面提供一种使用 **strings** 探索众多程序的简易方法。最重要的 Unix 实用工具都存放在两个目录中，这两个目录是 **/bin** 和 **/usr/bin**(我们将在第 23 章中对此展开讨论)。假设您希望查看这些目录中的一些程序。那么首先，输入下述两条 **cd**(change directory, 改变目录)命令中的一条。这将把“工作目录”改变到所选择的目录：

```
cd /bin
cd /usr/bin
```

现在使用 **ls**(list, 列举)程序显示目录中所有文件的清单：

```
ls
```

所有这些文件都是程序，使用 **strings** 可以查看它们。此外，因为这些文件位于工作目录中，所以不必指定完整的路径名。在这种情况下，指定文件名本身就足够了。例如，如果工作目录是 **/bin**，由于 **date** 程序就位于这个目录中，因此可以使用下述命令查看 **date** 程序的内部：

```
strings -7 date | sort -iu | less
```

通过这种方式，可以查看最重要的 Unix 实用工具内部的隐藏字符串。结束体验后，可以输入命令：

```
cd
```

这将把工作目录改变回 **home** 目录(第 23 章解释)。

19.16 转换字符：tr

相关过滤器：**sed**

tr(translate, 转换)程序可以对字符执行 3 种不同的运算。首先，它可以将字符改变成其他字符。例如，将小写字母改变成大写字母，或者将制表符改变成空格。或者，将每个数字“0”的实例改变成字母“X”。当这样做时，我们称之为**转换字符**。

其次，可以指定如果要转换的字符连续出现不止一次，则用一个单独的字符替换。例如，可以将一个或多个连续数字替换为字母“X”。或者，可以将多个空格替换为一个空格。当进行这样的改变时，我们称之为**挤压(squeeze)**字符。

最后，**tr**可以删除指定的字符。例如，可以删除文件中的所有制表符。或者，可以指定删除所有不是字母或数字的字符。

在接下来的几节中，将依次讨论各种运算。但是，在开始之前，我们首先示范 **tr** 程序的语法：

```
tr [-cds] [set1 [set2]]
```

其中 *set1* 和 *set2* 是字符组*。

注意该语法并不要求指定输入或输出的文件名。这是因为 **tr** 是一个纯过滤器，只从标准输入读取数据，仅向标准输出写入数据。如果希望从文件中读取数据，则必须重定向标准输入；如果希望将结果写入到文件中(保存结果)，则必须重定向标准输出。当看过后面的示例之后就会理解它们的含义(重定向在第 15 章中解释过)。

tr 程序执行的基本运算就是转换。在转换时需要指定两组字符。**tr** 读取数据后，它查找第一组数据。当 **tr** 查找到读取的字符时，它就使用第二组中相应的字符替换这些字符。例如，假设您有一个文件 **old**。您希望将所有的字符“a”替换为“A”。完成该任务的命令为：

```
tr a A < old
```

为了保存输出，只需将输出重定向到某个文件，例如：

```
tr a A < old > new
```

通过定义较长的字符组，可以同时替换多个字符。下述命令查找并进行 3 种不同的替换：“a”替换为“A”，“b”替换为“B”，以及“c”替换为“C”。

```
tr abc ABC < old > new
```

如果第二组字符比第一组字符少，那么第二组中最后一个字符就是重复的。例如，下述两条命令是等价的：

```
tr abcde Ax < old > new  
tr abcde Axxxx < old > new
```

这两条命令都将“a”替换为“A”，其他 4 个字符(“b”、“c”、“d”、“e”)替换为“x”。

当指定对 shell 有特殊含义的字符时，必须引用它们(参见第 13 章)，告诉 shell 按字面意义上的字符对待它们。尽管可以使用单引号或者双引号，但是大多数情况下，单引号最好。不过，如果只引用一个单独的字符，那么使用一个反斜线(参见第 13 章)更容易。

通常，最好能引用所有不是数字或字母的字符。例如，假设您希望将所有的冒号、分号和问号改变成点号，则可以使用：

```
tr ';;?' \. < old > new
```

tr 的强大缘于它能够处理字符范围。例如，考虑下述命令，该命令将所有的大写字母改变成小写字母(该命令是一条很长的命令)。

* 如果使用的是 Solaris，则应该使用伯克利 Unix 版本的 **tr** 程序。该程序存储在目录 **/usr/ucb** 中，因此要确保该目录位于搜索路径的开头(名称 **ucb** 代表 University of California, Berkeley，即加利福尼亚大学伯克利分校)。

我们在第 2 章中讨论了伯克利版本的 Unix，在第 13 章中讨论了搜索路径。


```
tr ABCDEFGHIJKLMNOPQRSTUVWXYZ
    abcdefghijklmnopqrstuvwxyz < old > new
```

如上所示，大写字母和小写字母之间的对应是明确的。但是，键入整个字母表两次比较烦人。作为替代，可以使用连字符(-)根据下述语法定义一个字符范围：

```
start-end
```

其中 *start* 是字符范围的第一个字符，*end* 是字符范围的最后一个字符。

例如，上述例子可以重写为：

```
tr A-Z a-z < old > new
```

范围可以是任意希望的字符组，只要它们在所使用的排序序列中形成一个连续的序列即可(排序序列在本章前面讨论过)。例如，下述命令实现一个密码，可以用来编码数值数据。数字 0 至数字 9 被字母表的前 10 个字母(即 A 到 J)分别替换。例如，375 被替换为 DHF。

```
tr 0-9 A-J < old > new
```

为了方便起见，还有几种缩写可以用来替代范围。这些缩写称为“预定义字符类”，我们将在第 20 章中讨论正则表达式时讨论它们。现在，只需知道可以使用[:lower:] 替换 a-z、[:upper:] 替换 A-Z 和[:digit:] 替换 0-9。例如，下述两条命令是等价的：

```
tr A-Z a-z < old > new
tr [:upper:] [:lower:] < old > new
```

下面这两条命令也是等价的：

```
tr 0-9 A-I < old > new
tr [:digit:] A-J < old > new
```

注意方括号和冒号是名称的一部分。

实际上，这 3 个预定义类都是 **tr** 程序最常使用的预定义类。但是，预定义类还有很多，第 20 章中的图 20-3 列举了预定义类的完整列表。

提示

与其他过滤器相比，**tr** 显得不平常，因为它不允许直接指定输入文件或输出文件的名称。为了从文件中读取数据，必须重定向标准输入；为了向文件中写入数据，必须重定向标准输出。基于这一原因，初学者在 **tr** 程序中最常见的错误就是没有进行重定向。例如，下述命令将无法运行：

```
tr abc ABC old
tr abc ABC old new
```

Linux 将显示一个含糊的消息，说明存在一个“额外的操作数”。其他类型的 Unix 将显示更没有帮助价值的消息。基于这一原因，可能有一天，您发现自己花费了大量的时间才明白 **tr** 程序工作不正常的问题。

解决方法永远不能忘记，当对 **tr** 使用文件时，总是需要进行重定向：

```
tr abc ABC < old
tr abc ABC < old > new
```

19.17 转换不可显示字符

到目前为止，所有的例子都相当简单。然而，它们还有点人为的痕迹。毕竟，生活中会有多少次需要将冒号、分号和问号改变成点号？或者将字母“abc”改变成“ABC”？或者使用密码将数字改变成字母？从传统上讲，**tr** 程序一直用于更深奥的转换，且通常包含不可显示字符。下面举一个典型的例子说明这一思想。

第 7 章解释过，在文本文件中，Unix 使用一个换行(^J)字符^{*}标记每一行的末尾，而 Windows 使用一个回车字符后跟一个换行(^M^J)字符标记每行的末尾。旧版本的 Macintosh 操作系统，一至到 OS 9，都使用回车(^M)字符^{**}标记每行的末尾。

假定您有一个文本文件，这个文件来自老版本的 Macintosh。在该文件中，每行末尾由一个回车字符标记。在 Unix 中使用这个文件之前，需要将所有的回车字符改变成换行字符。通过使用 **tr** 程序，这种情况就容易处理。但是，为了这样做，还需要一种同时表示换行和回车字符的方式。方法有两种。

第一种方法就是使用 **tr** 程序可以识别的特殊代码：**\r** 表示回车，**\n** 表示换行。另外，还可以使用一个反斜线(\)，后面跟一个代表该字符的 3 位八进制数。在这个例子中，**\015** 表示回车，**\012** 表示换行。出于参考目的，图 19-4 列举了 **tr** 通常使用的特殊代码以及相应的八进制值。

代码	控制键	八进制码	名称
\b	^H	\010	退格
\t	^I	\011	制表符
\n	^J	\012	换行/换行
\r	^M	\015	回车
\\	-	-	反斜线

图 19-4 tr 程序用来表示控制字符的代码

tr 程序用来将指定的字符转换(改变)成其他字符。为了指定不可显示字符，需要使用特殊代码，或者使用反斜线(\)后面跟该字符的 3 位八进制数(基 8)。附录 D 中的 ASCII 码列表列举了所有字符的八进制值。

本表列出了最常使用的 4 个控制字符的特殊代码和八进制值。**tr** 程序中还有其他一些特殊代码，但是不经常使用。

注意，因为反斜线用作转义字符(参见第 13 章)，所以如果希望指定一个实际的反斜线，必须连续使用两个反斜线。

^{*} 正如第 7 章讨论的，当 Unix 人士书写控制键的名称时，通常使用^作为“Ctrl”的缩写。因此，^J 指<Ctrl-J>。

^{**} 多年以来，Macintosh 操作系统(Mac OS)使用^M 标记文本行的末尾。正如前面所述，这种情况持续到 OS 9。2001 年，OS 9 被 OS X 所取代，而 OS X 基于 Unix。与其他基于 Unix 的系统相似，OS X 使用^J 标记文本行的末尾。

八进制值是字符的 ASCII 码对应的基 8^{*}数值。出于参考目的，附录 D 中列举了所有 ASCII 码的八进制制。

下面考虑如何使用 **tr** 将回车改变成新行。假设有一个文本文件 **macfile**，在这个文件中每行都以一个回车结尾。我们希望将所有的回车改变成新行，并将输出保存在文件 **unixfile** 中。下述两条命令都可以完成这个任务：

```
tr '\r' '\n' < macfile > unixfile
tr '\015' '\012' < macfile > unixfile
```

可以看出，一旦理解了特殊代码的原理，它们的使用将非常简单。例如，下述两行命令将 **olddata** 文件中所有的制表符都改变成空格，并将输出保存在 **newdata** 中^{**}：

```
tr '\t' ' ' < olddata > newdata
tr '\011' ' ' < olddata > newdata
```

19.18 转换字符：高级话题

到目前为止，我们已经讨论了如何使用 **tr** 进行直接的转换，在这种转换中，一个字符被替换为另一个字符。现在，我们准备讨论更高级的话题。在此之前，再次示范一次 **tr** 程序的语法：

```
tr [-cds] [set1 [set2]]
```

其中 **set1** 和 **set2** 是字符组。

-s 选项告诉 **tr** 第一组中的多个连续字符应该替换为一个单独的字符。正如前面所述，当这样做时，我们称之为挤压字符。下面举例说明。

下述两条命令将任意的数字(0-9)替换为大写字母“X”。输入来自文件 **olddata**，输出写入到文件 **newdata** 中：

```
tr [:digit:] X < olddata > newdata
tr 0-9 X < olddata > newdata
```

现在这些命令将每个数字都替换为“X”。例如，6 位数字 **120357** 将被替换为 **XXXXXX**。但是，假设您希望将所有的多位数字替换成一个单独的“X”，而不管数字有多长，那么您可以使用 **-s** 选项：

```
tr -s [:digit:] X < olddata > newdata
tr -s 0-9 X < olddata > newdata
```

^{*} 通常，我们采用基 10(十进制)系统计数，使用 10 个数字，从 0 至 9。但是，当使用计算机时，还有其他 3 种重要的计数系统：

- 基 2(二进制)：使用 2 个数字，0-1
- 基 8(八进制)：使用 8 个数字，0-7
- 基 16(十六进制)：使用 16 个数字：0-9 和 A-F

我们将在第 21 章中讨论计数系统。

^{**} 当将制表符改变成空格，或者将空格改变成制表符时，通常使用 **expand** 和 **unexpand**(参见第 18 章)。这两个程序是专门设计用来处理此类改变的，因此提供了更多的灵活性。

这将告诉 **tr** 将所有的大位数字挤压成一个单独的字符。例如，数字 **120357** 现在被修改为 **X**。

下面举一个有用的例子，在这个例子中我们挤压多个字符，但实际上并不改变字符。您希望将连续的空格替换为一个单独的空格。解决方法就是使用一个空格替换一个空格，同时挤压额外的空格：

```
tr -s ' ' < olddata > newdata
```

下一个选项是 **-d**，该选项删除指定的字符。这样，当使用 **-d** 选项时，只需定义一组字符。例如，为了删除所有的左圆括号和右圆括号，可以使用：

```
tr -d '()' < olddata > newdata
```

为了删除所有的数字，可以使用下面两条命令中的一个：

```
tr -d [:digit:] < olddata > newdata
tr -d 0-9 < olddata > newdata
```

最后一个选项是 **-c**，该选项最复杂，但是功能也最强大。该选项告诉 **tr** 匹配所有不在第一组中的字符*。例如，下述命令将除空格和新行字符之外的所有字符都替换为“X”：

```
tr -c ' \n' X < olddata > newdata
```

该命令的效果就是保留文本的“映像”，但是不保留文本的含义。例如，假设文件 **olddata** 包含：

```
"Do you really think you were designed to spend most of
your waking hours working in an office and going to
meetings?" - Harley Hahn
```

那么上述命令将生成：

```
XX XXX XXXXXX XXXXX XXX XXXX XXXXXXXX XX XXXXX XXXX XX
XXXX XXXXXX XXXXX XXXXXXXX XX XX XXXXXX XXX XXXXX XX
XXXXXXXXXX X XXXXXX XXXX
```

在结束 **tr** 的讨论之前，下面示范一个有趣的例子，在这个例子中，我们组合使用 **-c**(complement, 补数)和 **-s**(squeeze, 挤压)选项，统计使用的单词数量。假设您撰写了两篇历史论文，存储在文件 **greek** 和 **roman** 中。您希望统计两个文件中使用的单词的数量。策略如下所示：

- (1) 使用 **cat** 组合文件。
- (2) 使用 **tr** 将每个单词放在一个单独行上。
- (3) 使用 **sort** 对行进行排序，并消除重复行。

* 名称 **-c** 代表“complement, 补数”，它是一个数学术语。在集合论中，集合的补集指所有不在这个集合中的元素。例如，对于整数来说，偶数集合的补集就是所有奇数的集合。对于所有的大写字母来说，{ABCDWXYZ}集合的补集就是{EFGHIJKLMNOPQRSTU}集合。

(4) 使用 **wc** 统计剩余行的数量。

为了将每个单词放在一个单独行上(第2步)，我们只需使用 **tr** 将每个不属于单词的字符替换为一个新行字符。例如，假设有下列单词：

```
As you can see
```

它们将改变成：

```
As
you
can
see
```

为了使事情尽量简单，我们假设单词由 53 个不同字符组成：26 个大写字母、26 个小写字母以及撇号(也就是单引号)。下述 3 条命令都能完成这一任务——您可以选择一个自己喜欢的：

```
tr -cs [:alpha:]\' "\n"
tr -cs [:upper:][:lower:]\' "\n"
tr -cs A-Za-z\' "\n"
```

-c 选项改变不属于第一个集合的字符，**-s** 选项挤压重复字符。这些命令的效果就是将所有不是字母或者撇号的字符替换为新行字符。

一旦将单词隔离，即每行一个单词，那么剩下来的单词排序就简单了。只需使用 **sort** 程序和 **-u(unique, 唯一)** 选项消除重复行，同时使用 **-f(fold, 同等)** 选项忽略大写字母和小写字母之间的区别。然后再使用 **wc -l** 统计行的数量。下面是完整的管道线：

```
cat greek roman | tr -cs [:alpha:]\' "\n" | sort -fu | wc -l
```

更通用的表达方式为：

```
cat file1... | tr -cs [:alpha:]\' "\n" | sort -fu | wc -l
```

通过这种方式，一个简单的 Unix 管道线就可以统计一组输入文件中所包含的单词的数量。如果希望保存单词列表，则只需重定向 **sort** 程序的输出：

```
cat file1... | tr -cs [:alpha:]\' "\n" | sort -fu > file
```

19.19 非交互式文本编辑：sed

文本编辑器就是一个允许对文本行执行操作的程序。一般情况下，可以进行插入、删除、改变、搜索等操作。两个最重要的 Unix 文本编辑器是 **vi**(在第 22 章中讨论)和 **Emacs**。另外还有其他几个不重要，但是较简单的文本编辑器 **kedit**、**gedit**、**Pico** 和 **Nano**，这几个编辑器我们已经在第 14 章讨论过。

所有这些程序最常见的特征就是它们都是交互式的。也就是说，通过打开文件，然后

一条接一条地输入命令直到完成任务来使用这些程序。在本节中，将介绍一个文本编辑器，它就是 **sed**，这个文本编辑器是非交互式的。

对于非交互式文本编辑器来说，需要提前设计命令，然后将命令发送给程序，程序再自动地执行命令。通过使用非交互式文本编辑器，可以自动执行大量的任务，否则，只能手工执行。

sed 的使用方法有两种。第一种，让 **sed** 从文件中读取输入。这样将允许对已有文件自动地进行改变。例如，您希望读取一个文件，并将所有的“harley”修改为“Harley”。

第二种，使用 **sed** 在管道线中作为过滤器。这样就允许编辑程序的输出。另外，它还允许将 **sed** 的输出管道传送给另一个程序做进一步处理。

在开始之前，先解释几个术语。当考虑数据在管道线中从一个程序发送到另一个程序时，可以认为数据就像在管道中流动的水一样。基于这一原因，当数据从一个程序流向另一个程序时，我们称之为流。更具体地讲，当数据是由程序读取时，我们就称数据是输入流。当数据是由程序写入时，我们就称数据是输出流。

在我们讨论过的所有过滤器中，**sed** 程序是目前为止功能最强大的。这是因为 **sed** 并不是一个单用途的程序。它实际上是一个可移植的、与 shell 无关的语言解释器，被设计用来对数据流执行文本转换。因此 **sed** 就是“stream editor，流编辑器”的缩写。

对 **sed** 的所有方面进行全面的讨论已经超出了本书的范围。但是，**sed** 可以执行的最有用的操作就是进行简单的替换，因此我们将讲授这些内容。然而，我们省略了许多内容，因此，当您有空时，可以上网查看 **sed** 的指南，学习更多的知识。如果需要参考，还可以查看系统上的说明书页(**man sed**)。

以这种方式使用 **sed** 程序时的语法为：

```
sed [-i] command | -e command... [file...]
```

其中 *command* 是一个 **sed** 命令，*file* 是输入文件的名称。

为了展示如何使用 **sed** 程序，下面举一个典型的例子，在这个例子中，我们需要将每个“harley”都改变成“Harley”。其中输入来自文本文件 **names**，输出写入到文件 **newnames** 中：

```
sed 's/harley/Harley/g' names > newnames
```

稍后再解释实际命令的细节，现在先讨论一下输入文件和输出文件。

sed 程序从数据流中每次读取一行，按照下面 3 个步骤，从头至尾处理全部数据：

- (1) 从输入流中读取一行。
- (2) 执行指定的命令，对该行进行必要的变换。
- (3) 将该行写入到输出流中。

默认情况下，**sed** 将输出写入到标准输出，这意味着 **sed** 程序不改变输入文件。在一些情况下，这样很好，因为人们不希望改变原始文件，而是希望将输出重定向到另一个文件。从上面的例子中可以明白这一点。输入来自文件 **names**，而输出写入到文件 **newnames**。文件 **names** 的内容保持不变。

但是，大多数时候，人们确实希望改变原始文件。为了这样做，必须使用 **-i**(in-place，

原地)选项。这将导致 **sed** 程序将输出保存到一个临时文件。一旦所有的数据都成功处理完，**sed** 就将临时文件复制到原始文件。最终获得的效果就是改变了原始文件，但是前提是 **sed** 程序在执行任务的过程中没有出错。下面是一个使用 **-i** 选项的典型 **sed** 命令：

```
sed -i 's/harley/Harley/g' names
```

在这个例子中，**sed** 通过把所有的“harley”都改成“Harley”修改了输入文件 **names**。^{*}

当使用 **sed -i** 时，必须特别小心。因为对输入文件的改变是永久性的，而且没有任何“取消”命令。

提示

在使用 **sed** 改变文件之前，最好先运行一个没有 **-i** 选项的程序预览一下改变后的情况。例如：

```
sed 's/xx/XXX/g' file | less
```

这样可以查看输出，看看输出是不是期望的结果。如果是，则可以返回并使用有 **-i** 选项的命令进行改变^{**}：

```
sed -i 's/xx/XXX/g' file
```

19.20 使用 **sed** 进行替换

相关过滤器：**tr**

sed 的功能来自于可以让它执行的操作。其中最重要的操作就是替换，该操作使用的命令是 **s**。**s** 命令的语法有两种形式：

```
[/address|pattern/]s/search/replacement/[g]
```

其中 *address* 是输入流中一个或多个行的地址，*pattern* 是一个字符串，*search* 是正则表达式，*replacement* 是替换文本。

在最简单的形式中，通过指定一个搜索字符串和一个替换字符串使用替换命令。例如：

```
s/harley/Harley/
```

^{*} **-i** 选项只在 GNU 版本的 **sed** 程序中可用。如果系统没有使用 GNU 实用工具，例如 Solaris 系统，那么就不能使用 **-i** 选项。此时，为了使用 **sed** 改变文件，必须将输出保存到一个临时文件。然后使用 **cp**(copy, 复制)程序将临时文件复制到原始文件，并使用 **rm**(remove, 移除)程序将临时文件删除。例如：

```
sed 's/harley/Harley/g' names > temp
```

```
cp temp names
```

```
rm temp
```

换句话说，必须手动完成 **-i** 选项自动完成的功能。

^{**} 有一个通用的 Unix 原则，就是在进行重要的永久性改变之前，尽可能地事先预览结果。

我们在第 13 章讨论历史列表和别名时使用了一个相似的策略。这两处我们都讨论了在执行实际删除之前，如何通过预览结果而避免不小心删错文件。

该原则非常重要，希望您永远记住。

该命令告诉 **sed** 搜索输入流的每一行，查找字符串 “harley”，如果找到这个字符串，则将它改变成 “Harley”。默认情况下，**sed** 只改变每一行上第一个出现的匹配字符串。例如，假设下述行是输入流的一部分：

```
I like harley. harley is smart. harley is great.
```

那么上述命令将把这一行改变成：

```
I like Harley. harley is smart. harley is great.
```

如果希望改变所有的匹配字符串，则需要在命令的末尾键入后缀 **g**，以代表 “global，全局”：

```
s/harley/Harley/g
```

在我们的例子中，添加 **g** 后将使原始行被改变成：

```
I like Harley. Harley is smart. Harley is great.
```

依我的个人经验看，当使用 **sed** 进行替换时，通常希望使用 **g** 改变所有匹配字符串，而不是每一行中的第一个匹配字符串。这就是我们的所有例子中都包含 **g** 后缀的原因。

到目前为止，我们只搜索了简单的字符串。实际上，通过使用所谓的 “正则表达式” (通常简称为 “**regex**”) 可以进行更强大的搜索。使用正则表达式将允许指定模式，从而提供更大的灵活性。但是，正则表达式比较复杂，需要花一定的时间学习如何使用它们。

现在我们还打算讨论正则表达式使用上的细节问题。实际上，它们非常复杂，而且功能也非常强大，所以我们专门提供一章也就是第 20 章来讨论正则表达式。一旦您阅读了这一章内容之后，我希望您再回到本节来，花一些时间体验正则表达式和 **sed** (一定要使用图 20-1、20-2 和 20-3 中的参考手册)。

现在，我们示范两个对 **sed** 使用正则表达式的例子。首先，假设您有一个文件 **calendar**，该文件中包含下几个月的计划信息。您希望将所有出现的字符串 “mon” 或 “Mon” 改变成单词 “Monday”。下面就是一条使用正则表达式进行这个改变的命令：

```
sed -i 's/[mM]on/Monday/g' calendar
```

为了理解该命令，只需知道，在正则表达式中，记号 [...] 匹配方括号中的任何单个字符，在这个例子中，或者是 “m”，或者是 “M”。因此，搜索字符串或者是 “mon”，或者是 “Mon”。

第二个例子需要一点技巧。在本章前面，当讨论 **tr** 程序时，我们讨论了 Unix、Windows 和 Macintosh 如何使用不同的字符标记文本行的末尾。Unix 使用一个换行 (^J) 字符，Windows 使用一个回车后跟一个换行 (^M^J) 字符，而 Macintosh 使用一个回车 (^M) 字符 (这些字符的详细讨论请参见第 7 章)。

在讨论期间，我示范了如何将一个 Macintosh 格式的文本文件转换成一个 Unix 格式的文本文件。使用 **tr** 将所有的回车改变成换行字符即可完成该任务，命令如下：

```
tr '\r' '\n' < macfile > unixfile
```

但是，如果有一个 Windows 格式的文本文件，那么如何在 Unix 中使用它呢？换句话说，如何将每个文本行末尾的“回车新行”字符改变成一个简单的新行字符呢？这时不能使用 **tr**，因为需要将两个字符(^M^J)改成一个字符(^J)，**tr** 只能将一个字符改变成另一个字符。

但是，我们可以使用 **sed**，因为 **sed** 可以改变任何内容。为了创建该命令，我们利用回车字符(^M)位于行的末尾，且正好在新行字符(^J)之前这一事实。我们只需查找并删除 ^M 即可。

下面是完成该转换的两条命令。第一条命令从文件 **winfile** 中读取输入，并将输出写入到文件 **unixfile** 中。第二条命令使用 **-i** 选项改变原始文件本身：

```
sed 's/.$//' winfile > unixfile
sed -i 's/.$//' winfile
```

那么它的原理是什么呢？在正则表达式中，.(点)字符匹配任何单独的字符，而**\$**(美元符号)字符匹配行的末尾。因此，搜索字符串**.\$**就是指新行字符前面的字符。

仔细地看看替换字符串，注意它是空的。这告诉 **sed** 将搜索字符串改变成空串。也就是说，告诉 **sed** 删除搜索字符串。这样获得的结果就是将文件中每行末的回车字符移除。

如果您以前没有使用过正则表达式，那么我并不期望您能够完全理解最后几条命令。但是，我向您保证，当阅读完第 20 章内容时，这些例子以及其他类似的例子，都将非常容易理解。

提示

为了使用 **sed** 删除字符串，只需搜索该字符串，并将它替换为空即可。

这是一个需要记住的重要技术，因为在允许搜索和替换操作的程序中都可以使用这一技术(实际上，在文本编辑器中经常使用该技术)。

19.21 告诉 sed 只对指定行进行操作

默认情况下，**sed** 对数据流中的每一行都执行它的操作。为了改变这一点，可以在命令前面加一个“地址”。这将告诉 **sed** 只对拥有该地址的行进行操作。地址的语法如下所示：

```
number[,number] | /regex/
```

其中 *number* 是行号，*regex* 是正则表达式。

在最简单的形式中，地址就是一个单独的行号。例如，下述命令只改变数据流的第 5 行：

```
sed '5s/harley/Harley/g' names
```

为了指定行的范围，可以使用逗号将两个行号分开。例如，下述命令改变第 5 行至第 10 行：

```
sed '5,10s/harley/Harley/g' names
```

为了方便起见，可以使用字符\$(美元符号)指定数据流的最后一行。例如，为了改变数据流的最后一行，可以使用：

```
sed '$s/harley/Harley/g' names
```

为了改变第 5 行至最后一行，可以使用：

```
sed '5,$s/harley/Harley/g' names
```

另一种指定行号的方法就是使用正则表达式或者被/(斜线)字符包围的字符串*。这告诉 sed 只处理那些包含指定模式的行。例如，为了只改变那些包含字符串“OK”的行，可以使用：

```
sed '/OK/s/harley/Harley/g' names
```

下面举一个更复杂的例子。下述命令只改变那些包含两个连续数字的行：

```
sed '/[0-9][0-9]/s/harley/Harley/g' names
```

(表示法[0-9]指从 0 至 9 的任意一个单独数字，细节请参见第 20 章。)

19.22 使用非常长的 sed 命令

正如前面所述，sed 实际上是一种文本操作编程语言的解释器。因此，可以编写包含任意多个 sed 命令的程序，并将程序存储在文件中，每当需要时就可以运行。

为了这样做，需要使用-f 命令标识程序文件。例如，为了运行存储在文件 instructions 中的 sed 程序，并使用文件 input 中的数据，可以使用命令：

```
sed -f instructions input
```

使用 sed 编写程序已经超出了本书的范围。在本章中，我们只关注如何将 sed 作为过滤器使用。然而，有时候也希望 sed 执行几个操作，这实际上就是运行一个微小的程序。当需要这样做时，可以指定任意多的 sed 命令，只要每个 sed 命令之前使用一个-e(editing command, 编辑命令)选项即可。下面举例说明。

假设您有一个文件 calendar，其中存放的是一个日程表。在这个文件中，有不同的缩写需要扩展。特别地，您希望将“mon”改成“Monday”。使用的命令为：

```
sed -i 's/mon/Monday/g' calendar
```

但是，您也希望将“tuc”改变成“Tuesday”。这就需要两条单独的 sed 命令，且两条 sed 命令前面都必须加一个-e 选项：

* 正如第 20 章中将要讨论的，字符串将被当作正则表达式。

```
sed -i -e 's/mon/Monday/g' -e 's/tue/Tuesday/g' calendar
```

现在，您可以看出模式了。您需要 7 条单独的 **sed** 命令，一条命令代表一星期的一天。但是，这要求一个非常长的命令行。

正如第 13 章中讨论的，输入长命令的最好方式就是将命令分隔成多行。为此，只需在按<Return>键之前，键入一个\（反斜线）字符。反斜线引用新行字符，从而允许将命令分隔在多行上。

作为一个例子，下面是一个长的 **sed** 命令，用来修改一星期 7 天的缩写。注意所有行（除最后一行外）都是连续的。实际上，在这里看到的是一个非常长的命令行：

```
sed -i \  
-e 's/mon/Monday/g' \  
-e 's/tue/Tuesday/g' \  
-e 's/wed/Wednesday/g' \  
-e 's/thu/Thursday/g' \  
-e 's/fri/Friday/g' \  
-e 's/sat/Saturday/g' \  
-e 's/sun/Sunday/g' \  
calendar
```

提示

当键入\<Return>以继续一行时，大多数 shell 都显示一个特殊的提示，称为辅助提示（secondary prompt），指示命令还需要继续。

在 Bourne Shell 家族（**Bash**、**Korn Shell**）中，默认的辅助提示是一个>（大于号）字符。通过修改 shell 变量 **PS2** 可以修改辅助提示（但是大多数人不会这样做）。

在 **C-Shell** 家族中，只有 **Tcsh** 有辅助提示。默认情况下，它是一个？（问号）。通过修改 shell 变量 **prompt2** 可以修改辅助提示。

（修改 shell 变量的命令在第 12 章解释过，将这样的命令放在初始化文件中在第 14 章讨论过。）

19.23 练习

1. 复习题

1. 在所有的过滤器中，**grep** 是最重要的过滤器。**grep** 有何用处？为什么它在管道线中特别有用？解释下述选项的含义：**-c**、**-i**、**-l**、**-L**、**-n**、**-r**、**-s**、**-v**、**-w** 和 **-x**。
2. **sort** 程序可以执行的两项任务各是什么？解释下述选项的含义：**-d**、**-f**、**-n**、**-o**、**-r** 和 **-u**。为什么 **-o** 选项是必需的？
3. 什么是排序序列？什么是区域设置？是什么将它们连接在一起？
4. **uniq** 程序可以执行哪 4 项任务？

5. **tr** 程序可以执行哪 3 项任务？当使用 **tr** 时，使用什么特殊码表示下述字符：退格、制表符、新行/换行、回车和反斜线？

2. 应用题

1. 正如第 23 章将要讨论的，**/etc** 目录用来存放配置文件(第 6 章解释)。创建一条命令，浏览**/etc** 目录中的所有文件，搜索包含单词“**root**”的所有行。输出应该每次一屏地显示。提示：为了指定文件名，可以使用模式**/etc/***。

在搜索**/etc** 目录中的文件的过程中将生成几个假错误消息。创建第二条命令，将所有这样的消息抑制掉。

2. 有人打赌，不使用字典，您无法找到 5 个以字符串“**book**”开头的英语单词。不过，您被允许使用一条 Unix 命令。那么，应该使用哪一条命令呢？

3. 您为学校运行一个在线约会服务。您有 3 个文件，分别是 **reg1**、**reg2** 和 **reg3**，这 3 个文件包含的是用户注册信息。在这些文件中，每行包含一个人的信息。

创建一个管道线处理所有 3 个文件，只选取那些包含单词“**female**”或“**male**”(自己决定)的行。在消除所有的重复行之后，结果应该保存在文件 **prospects** 中。

一旦完成这些，创建第二个管道线，显示所有注册过不止一次的人的列表(**male** 或 **female**)。提示：在这些文件中查找重复行。

4. 您有一个文本文件 **data**。创建一个管道线，显示所有双单词(**double word**)的实例，例如“**hello hello**”(假定一个“单词”由连续的大写字母或小写字母构成)。提示：首先创建一个所有单词的列表，每行一个单词。然后将输出管道传送给搜索连续相同行的程序。

3. 思考题

1. 在前面的一个问题中，我提到 **grep** 是最重要的过滤器，而且还要求解释为什么它在管道线中特别有用。您认为该问题的答案应该是什么呢？**grep** 看上去如此强大和有用，反应了人的什么特性？

2. 最初，Unix 基于美国英语和美国数据处理标准(例如 ASCII 码)。随着国际化工具和标准(例如区域设置)的开发，Unix 现在可以由来自许多不同文化的人使用。这样的用户能够以他们自己的语言，使用他们自己的数据处理习惯与 Unix 交互。通过这种方式扩展 Unix 需要有哪些权衡考虑呢？分别列举它的 3 个优点和 3 个缺点。

3. 在本章中，我们详细讨论了 **tr** 和 **sed** 程序。可以看出，这两个程序都非常有用。但是，它们都是复杂的工具，要求大量的时间和精力去掌握。对于一些人来说，这不是问题。但是，对于大多数人来说，花大量的时间去学习如何很好地使用一个复杂的工具并不是一种舒服的体验。您认为为什么会是这样呢？是不是所有的工具都应该设计为容易学习和使用呢？

4. 对下述命题进行评价：整个 Unix 工具箱中的所有程序在掌握过程中所花的时间都比学习如何熟练地弹钢琴所花的时间要少。

正则表达式

正则表达式用来指定字符串的模式，其最常见的应用就是搜索字符串。因此，正则表达式经常用于搜索-替换操作。

作为一个工具，正则表达式如此有用，并且功能如此强大，因此经常可以看到整本书和整个网站专门来讨论这个话题。当然，掌握正则表达式的使用艺术是成为 Unix 专家的必备条件之一。

有时候，可能会创建非常复杂的正则表达式。但是，大多数时候，所需要的正则表达式都比较简单和直观，因此在学习正则表达式时，只需学习一些简单的规则，然后不断地练习、练习、再练习。本章的目标就是提供一个入门介绍。

在开始之前，我希望先提一件事情。在本章中，我将示范许多使用 **grep** 命令(参见第 19 章)的例子。如果您发现某些正则表达式的特性不适合自己的 **grep**，则可以试着使用 **egrep** 或 **grep -E**。在这种情况下，可以设置一个别名自动地使用这些变体。第 19 章对此进行了详细的阐述，参见有关 **grep** 和 **egrep** 的讨论。当在本章后面讨论扩展和基本正则表达式时，原因将会得到展示。

20.1 正则表达式简介

在本章中，将示范一些使用正则表达式的 **grep** 命令，**grep** 命令在第 19 章中讨论过。尽管这些例子都是一些简单的例子，但是您应该知道正则表达式可以在许多不同的 Unix 程序中应用，例如 **vi** 和 **Emacs** 文本编辑器、**less**、**sed** 等。另外，正则表达式还可以用于许多编程语言中，例如 **Awk**、**C**、**C++**、**C#**、**Java**、**Perl**、**PHP**、**Python**、**Ruby**、**Tcl** 和 **VB.NET**。

我将讲授的概念都是典型的正则表达式通用概念。一旦掌握了这些基本规则，那么当需要时，您只需学习少数几种变体即可。尽管正则表达式更深奥的特性可能在各个程序之间有所不同，例如 **Perl** 就有自己的一组高级特性，但是基本思想总是相同的。如果遇到了问题，只需查看所使用程序的文档资料即可。

正则表达式(regular expression)通常简写为 **regex** 或 **re**，是一种指定字符串模式的简洁方式。例如，考虑下面由 3 个字符串组成的一组字符串：

harley1 harley2 harley3

作为正则表达式，可以使用 **harley[123]**表示这组模式。

下面再举一个例子。您希望描述一组包含大写字母“H”，后面跟任意数量的小写字母，最后是小写字母“y”的字符串，那么所使用的正则表达式是 **H[[:lower:]]*y**。

可以看出，正则表达式的强大来自拥有特殊含义的元字符和缩写的使用。我们将在下几节中对此进行详细讨论。出于参考目的，图 20-1、20-2 和 20-3 中汇总了正则表达式的使用语法。现在请花一点时间浏览一遍，在阅读本章剩余内容的过程中，如果有需要，再返回来查看它们。

元字符	含义
.	除新行字符外，匹配任意的单个字符
^	锚：匹配行的开头
\$	锚：匹配行的末尾
\<	锚：匹配单词的开头
\>	锚：匹配单词的末尾
[list]	字符类：匹配 list 中的任何字符
[^list]	字符类：匹配不在 list 中的任何字符
()	组：视为一个单独的单元
	交变：匹配选择之一
\	引用：从字面上解释元字符

图 20-1 正则表达式：基本匹配

正则表达式是一种指定字符模式的简洁方式。在正则表达式中，普通字符匹配自身，特定的元字符拥有特殊的含义。本表列举了用来执行基本模式匹配功能的元字符。

运算符	含义
*	匹配 0 次或多次
+	匹配 1 次或多次
?	匹配 0 次或 1 次
{n}	限定：匹配 n 次
{n,}	限定：最少匹配 n 次
{0,m}	限定：最多匹配 m 次
{,m}	限定：最多匹配 m 次
{n,m}	限定：最少匹配 n 次，最多匹配 m 次

图 20-2 正则表达式：重复运算符

在正则表达式中，下述元字符称为重复运算符，可以用来匹配多个指定字符的实例。

注意：一些程序不支持 {,m}，因为它不是标准的。

类	含义	类似于
[lower:]	小写字母	a-z
[upper:]	大写字母	A-Z
[alpha:]	大小写字母	A-Za-z
[alnum:]	大小写字母、数字	A-Za-z0-9
[digit:]	数字	0-9
[punct:]	标点符号	—
[blank:]	空格或制表符(空白符)	—

图 20-3 正则表达式：预定义字符类

正则表达式中可以包含有定义一组字符的字符类。为了方便起见，正则表达式中有一组预定义字符类，可以将这些预定义字符类用作字符集合的缩写。本表示范了最重要的预定义字符类。注意方括号和冒号都是名称的一部分。

最右边的一列给出了与预定义字符类等价的字符范围。如果系统使用的是 C 排序序列，则可以使用这些范围取代类的名称。为了确保系统使用的是 C 排序序列，可以将环境变量 `LC_COLLATE` 的值设置为 C(详情请参见正文)。

20.2 正则表达式的起源

术语“正则表达式”来源于计算机科学，指的是一组指定模式的规则。该名称来源于著名的美国数学家和计算机科学家 Stephen Kleene(1909-1994)的工作(他的姓名发音为“Klej-nee”)。

在 20 世纪 40 年代初期，两名神经学科学家 Walter Pitts 和 Warren McCulloch 按照他们对神经元(神经细胞)工作方式的理解，开发了一个数学模型。作为模型的一部分，他们使用了非常简单的虚构机器，称为自动机(automata)。20 世纪 50 年代中期，Kleene 开发了一种描述自动机的数学方法，即使用所谓的“正则集”：能够用少量简单属性描述的集合。然后 Kleene 创建了一种简单的记号，用来描述这样的集合，他称这些记号为正则表达式。

1966 年，Ken Thompson，也即后来开发 Unix 系统的人，加入到贝尔实验室中来。他所做的首批事情之一就是编写一个新版本的 QED 文本编辑器。他在加利福尼亚大学伯克利分校曾经使用过 QED 文本编辑器。Thompson 极大地扩展了 QED，在他添加的众多功能中，包括一个使用 Kleene 正则表达式的增强形式的模式匹配功能。在此之前，文本编辑器只能搜索字符串。现在，通过使用正则表达式，QED 编辑器也可以进行模式搜索。

1969 年，Thompson 创建了第一版，即原始版本的 Unix(参见第 1 章和第 2 章)。不久之后，他编写了第一个 Unix 编辑器 **ed**(发音为“ee-dee”)，在这个编辑器中，他使用了一种简化形式的正则表达式，其功能要比在 QED 中使用的正则表达式的功能弱一些。**ed** 程序在 1971 年发布的 Unix Version 1 中提供。

ed 程序的流行导致了在 **grep** 中使用正则表达式，后来，许多其他 Unix 程序也开始使

用正则表达式。现在，正则表达式应用广泛，不仅仅是在 Unix 中，而且已经遍布计算领域(更有趣的是，Thompson 编写 **ed** 程序时省略的功能现在已经添加回来)。

提示

下一次参加 Unix 聚会，当人们讨论正则表达式时，您可以说正则表达式与乔姆斯基层次结构(Chomsky hierarchy)中的 Type 3 语法对应。一旦引起了人们的注意，接下来您就可以解释说在任何正则表达式和 NFA(nondeterministic finite automaton, 非确定性有限自动机)之间构建一个简单的映射关系是可行的，因为每个正则表达式都拥有有限的长度，因此，也就拥有有限数量的操作符。

不一会儿，您就会成为房间内最受欢迎的人*。

20.3 基本和扩展正则表达式

本节只是一个参考，首次阅读本节内容可能会觉得多少有点混乱。但是不要担心，稍后，当阅读完本章其他内容并经过练习之后，您就会理解本节所有内容。

正如前一节所述，当 Ken Thompson 创建 **ed** 文本编辑器时，正则表达式成为 Unix 的一部分。**ed** 文本编辑器随 Unix Version 1 于 1971 年发布。原始的正则表达式功能非常有用，但是也很有限，多年以来，它一直在不断地扩展。这使得各个正则表达式系统的复杂度不同，容易使初学者混淆。

实际上，您只需记住几个基本思想。现在我们讨论一下这些思想，这样在稍后练习本章中的例子时您就不会遇到麻烦。

Unix 支持两种主要的正则表达式变体：一个现代版本，一个以前的废弃版本。现代版本的正则表达式是扩展正则表达式(extended regular expression)，或者简称为 ERE。它是当前的标准，属于 IEEE 1003.2 标准(POSIX 的一部分，参见第 11 章)。

以前版本的正则表达式是基本正则表达式(basic regular expression)，或者简称为 BRE。它是多年以前使用的更原始类型的正则表达式，后来被 1003.2 标准取代。基本正则表达式要比扩展正则表达式的功能弱，而且语法也容易混淆。基于这些原因，现在已经将 BRE 废弃，保留它们只是为了与旧程序兼容。

在本章中，我们将讨论扩展正则表达式，也即现代 Unix 和 Linux 系统的默认正则表达式。但是，有时候，也有可能遇到一些旧程序，它们只接受基本正则表达式。在这种情况下，我希望您知道应该做什么，因此下面解释一下。

两条极有可能遇到问题的命令就是 **grep** 和 **sed**(这两条命令都在第 19 章中讨论过)。为了查看系统上的这两条命令支持什么类型的正则表达式，可以查看说明书页：

```
man grep
man sed
```

* 是的，我经常经历这种情况。

如果系统使用的是 GNU 实用工具，例如 Linux 和 FreeBSD 系统，那么一些命令已经升级为提供-E 选项，从而允许使用扩展正则表达式。例如，grep 就是这种情况。通常，可以通过查看命令的说明书页，或者使用--help 选项显示命令的语法(第 10 章讨论过)，以查看命令是否提供-E 选项。

提示

对于 Linux 和 FreeBSD 系统来说，一些命令提供-E 选项，允许使用扩展正则表达式。因为扩展正则表达式更可取，所以应该养成使用-E 的习惯。

如果经常使用这样的命令，则可以创建一个别名，强制自动地使用-E 选项。这方面的例子，请参见第 19 章中有关 grep 和 egrep 的讨论。

尽管扩展正则表达式是现代标准，同时一些程序提供了-E 选项，但仍然有时候不得不选择使用基本正则表达式。例如，您可能希望使用一个自己系统上的旧程序，而该程序只支持基本正则表达式(这种情况最常见的例子就是 sed)。

在这种情况下，您应该知道要做什么，因此我们先解释一下基本正则表达式和扩展正则表达式之间的区别。当然，该讨论还有点太早，因为我们还没有讨论正则表达式的技术细节。但是，正如前面所述，如果第一次阅读时无法理解，可以稍后再阅读一遍。

正如本章前面所述，正则表达式的强大功能来源于使用拥有特殊含义的元字符。我们将在本章中用大量的时间讨论如何使用这些元字符(出于参考目的，前面已将这些元字符汇总在图 20-1、20-2 和 20-3 中)。

基本正则表达式和扩展正则表达式之间的主要区别就是，对于基本正则表达式来说，有一些特定的元字符不能使用，而其他元字符必须使用反斜线引用(引用在第 13 章中讨论过)。不能使用的元字符有问号、加号和竖线：

? + |

必须转义的元字符有花括号和圆括号：

{ } ()

出于参考目的，图 20-4 中总结了这些限制。第一次阅读时可能并不能理解，但是当读完本章内容时您一定会理解其含义。

扩展正则表达式	基本正则表达式	含义
{ }	\{ \}	定义一个限定(花括号)
()	\(\)	定义一个组(圆括号)
?	\{0,1\}	匹配 0 次或 1 次
+	\{1,\}	匹配 1 次或多次
	—	交变：匹配选项中的一个
[[:name:]]	—	预定义字符类

图 20-4 扩展正则表达式和基本正则表达式

正则表达式的现代标准就是扩展正则表达式(ERE)，它作为 1003.2 POSIX 标准的一部分被定义。扩展

正则表达式取代了以前的基本正则表达式(BRE)。只要可以选择,就应该使用 ERE。

出于参考目的,本表列举了基本正则表达式的限制,可以总结如下:

- (1) 花括号必须使用反斜线引用。
- (2) 圆括号必须使用反斜线引用。
- (3) 不能使用?,但是可以使用`\{0,1\}`模拟。
- (4) 不能使用+,但是可以使用`\{1,\}`模拟。
- (5) 不能使用|(竖线)。
- (6) 不能使用预定义字符类。

20.4 匹配行和单词

正如前面所述,正则表达式(或者 `regex`)是一种指定字符模式的简洁方式。为了创建正则表达式,需要根据特定的规则将普通字符和元字符组合在一起,然后使用该正则表达式搜索希望查找的字符串。

当正则表达式对应于一个特定的字符串时,我们就说它匹配该字符串。例如,正则表达式 `harley[123]` 可以匹配 `harley1`、`harley2` 或者 `harley3`。现在,您还不用考虑细节问题,只需意识到 `harley` 和 `123` 都是普通字符,而 `[`和`]`都是元字符。另一种说法就是,在正则表达式中,`harley` 和 `123` 匹配自身,而 `[`和`]`(方括号)字符拥有特殊的含义。最终,您将会学习所有的元字符以及它们的特殊含义。

在本节中,我们将讨论特定的元字符,这些元字符称为锚(anchor),用来匹配在字符串的开头或末尾的位置。例如,正则表达式 `harley$` 匹配字符串 `harley`,但是 `harley` 只能位于行的末尾。这是因为 `$` 是一个元字符,它通过匹配行的末尾来充当锚(现在还不用关心细节问题)。

为了开始对正则表达式的研究,我们首先介绍基本规则:所有的普通字符,例如字母和数字,都与自身匹配。下面举几个例子,示范这一规则。

假设您有一个文件 `data`,该文件中包含下述 4 行内容:

```
Harley is smart
Harley
I like Harley
the dog likes the cat
```

您希望使用 `grep` 查看所有包含有“Harley”的行。使用的命令如下:

```
grep Harley data
```

在这个例子中,`Harley` 实际上是一个正则表达式,它将使 `grep` 选取第 1、2 和 3 行,但是不选取第 4 行:

```
Harley is smart
Harley
```



```
I like Harley
```

这没什么新鲜的，但是它确实示范了在正则表达式中，一个 **H** 匹配一个 “H”，一个 **a** 匹配一个 “a”，一个 **r** 匹配一个 “r”，等等。所有的正则表达式都派生于这一基本思想。

为了扩展正则表达式的功能，可以使用锚指定所查找模式的位置。**^**(音调符号)元字符是一个匹配行的开头的锚。因此，为了搜索那些以 “Harley” 开头的行，可以使用：

```
grep '^Harley' data
```

在我们的例子中，该命令将选取第 1 行和第 2 行，但是不选取第 3 行和第 4 行(因为它们不是以 “Harley” 开头的)：

```
Harley is smart
Harley
```

请注意，在上一条命令中，我引用了正则表达式。当使用包含元字符的正则表达式时，应该这样做以确保 shell 不理睬这些字符，将它们传递给程序(在这个例子中就是 **grep**)。如果不确定是否需要引用正则表达式，那就尽管引用，多余的引用并不会导致问题。

请注意，为了确保安全，我使用了强引用(单引号)，而不是弱引用(双引号)。这将确保所有的元字符，而不是其中的一部分，被正确地引用(如果需要回顾强引用和弱引用之间的区别，可以参见第 13 章)。

匹配行的末尾的锚是 **\$**(美元符号)字符。例如，为了搜索那些以 “Harley” 结尾的行，可以使用：

```
grep 'Harley$' data
```

在我们的例子中，这将只选取第 2 行和第 3 行：

```
Harley
I like Harley
```

^和**\$**可以在同一个正则表达式中组合使用，只要所描述的情形有意义即可。例如，为了搜索整行就是一个单词 “Harley” 的行，可以同时使用这两个锚：

```
grep '^Harley$' data
```

在我们的例子中，这将选取第 2 行：

```
Harley
```

通过使用这两个锚，但是两个锚之间不指定任何内容可以方便地查找空行。例如，下述命令统计文件 **data** 中的空行数量：

```
grep '^$' data | wc -l
```

以相似的方式，还可以使用锚来匹配单词的开头或末尾，或者两者都进行匹配。为了匹配一个单词的开头，可以使用两个字符的组合 **\<**。为了匹配单词的末尾，可以使用 **\>**。

例如，假设您希望搜索文件 **data**，查找所有包含字符串 “kn” 的行，但是字符串 “kn”

只能出现在单词的开头，则可以使用：

```
grep '\<kn' data
```

查找字符串“ow”，但是字符串“ow”只能出现在单词的末尾，可以使用：

```
grep 'ow\>' data
```

为了搜索完整的单词，可以同时使用\<和\>。例如，为了搜索“know”，并且“know”必须是一个完整的单词，可以使用：

```
grep '\<know\>' data
```

该命令将选取这样的行：

```
I know who you are, and I saw what you did.
```

但是它不选取下面这样的行：

```
Who knows what evil lurks in the hearts of men?
```

为了方便起见，在使用 GNU 实用工具的系统上，例如 Linux 和 FreeBSD 系统，可以使用**\b** 作为\<和\>的替代锚。例如，下述命令是等价的：

```
grep '\<know\>' data
```

```
grep '\bknow\b' data
```

可以认为**\b** 是“边界标记(boundary marker)”的含义。

现在您可以选择自己喜欢哪一种单词边界锚了。我使用\<和\>，因为对我的眼睛来说，它们比较直观。但是，许多人倾向于使用**\b**，因为它容易键入，而且可以在单词头和单词尾使用同一个锚。

当使用正则表达式时，“单词”的定义要比英语中的定义更灵活。在正则表达式中，单词就是一个自包含的，由字母、数字或者_(下划线)字符构成的连续字符序列。因此，在正则表达式中，下面的示例都是单词：

```
fussbudget Weedly 1952 error_code_5
```

许多 Unix 程序也都采用相同的定义。例如，当使用**-w** 选项匹配完整单词时，**grep** 使用的就是该定义。

提示

当使用 **grep** 查找完整单词时，使用**-w**(word, 单词)选项通常要比使用多个\<和\>或者**\b** 实例简单。例如，下述 3 条命令都是等价的：

```
grep -w 'cat' data
```

```
grep '\<cat\>' data
```

```
grep '\bcata\b' data
```

20.5 匹配字符；字符类

在正则表达式中，元字符.(点号)匹配任何单个的字符，新行字符除外(在 Unix 中，新行字符标记一行的末尾，参见第7章)。

例如，假设您希望搜索文件 **data**，查找所有包含下述模式的行：字符串“Har”，后面跟两个任意字符，再后跟一个字母“y”。可以使用下述命令：

```
grep 'Har..y' data
```

该命令将查找包含以下单词的行，例如：

```
harley harxxy harlly har12y
```

以后您将发现元字符.非常有用，而且还会大量地使用它。然而，有时候，您希望更具体一点：.匹配的是任意的字符，而您希望匹配特定的字符。例如，您可能希望搜索一个大写字母“H”，后面或者跟“a”，或者跟“A”。在这种情形下，可以通过将字符放在方括号[]中来指定希望搜索的字符。这样的结构就称为一个**字符类**。

例如，为了在文件 **data** 中搜索所有包含字母“H”、后面跟“a”或者“A”的行，可以使用：

```
grep 'H[aA]' data
```

在继续之前，我希望申明一个重点。严格地讲，字符类不包括方括号。例如，在上一条命令中，字符类是 **aA**，而不是**[aA]**。尽管在使用字符类时方括号是必须的，但是并不认为它们是字符类本身的一部分。当我们讨论“预定义字符类”时，这一区别非常重要。

下面举一个例子，在同一个的正则表达式中使用不止一个字符类。下面的这条命令搜索包含单词“license”的行，即便由于混用“c”和“s”而引起错误拼写的单词也可以搜索出来：

```
grep 'li[cs]en[cs]e' data
```

下面示范一个更有用的命令，该命令使用\<和\>或者\b只匹配整个单词：

```
grep '\<li[cs]en[cs]e\>' data
```

```
grep '\bli[cs]en[cs]e\b' data
```

这两条命令将匹配下面的任一单词：

```
licence license lisence lisense
```

20.6 预定义字符类；范围

有一些字符集是比较常见的，因此它们被冠以相应的名称，从而方便使用。这些字符集称为**预定义字符类**，如本章前面的图 20-3 所示(在继续之前，请再回过头去看一看，因

为我希望您熟悉各种名称及其含义)。

预定义字符类的使用比较直接，除了一个规则之外，这个规则是：方括号实际上是名称的一部分。因此，当使用预定义字符类时，必须包含第二组方括号，以维持正确的语法(前面讲过，当使用字符类时，外面的方括号不属于类)。

例如，下述命令使用 **grep** 查找文件 **data** 中包含数字 21，后面跟一个小写字母或大写字母的所有行：

```
grep '21[[:alpha:]]' data
```

下面的命令查找包含两个连续大写字母，后面跟一个数字，再跟一个小写字母的所有行：

```
grep '[[[:upper:]]][[:upper:]][[:digit:]][[:lower:]]' data
```

除了预定义字符类外，还有另一种方式指定一组字母或数字，即使用字符范围，具体方法是将第一个字符和最后一个字符用连字符分开。例如，为了搜索文件 **data** 中所有包含数字 3 至 7 的行，可以使用：

```
grep '[3-7]' data
```

范围 0-9 的含义同 `[[:digit:]]`。例如，为了搜索包含大写字母 “X”，后面跟任意两个数字的行，可以使用下面两条命令中的一个：

```
grep 'X[0-9][0-9]' data
grep 'X[[:digit:]][[:digit:]]' data
```

下面再考虑一种情形：匹配不在特定字符类之中的字符。为此，只需在开头的左方括号之后放一个 `^`(音调符号)即可。在这种情况下，`^` 充当一个否定操作符。

例如，下述命令搜索文件 **data**，查找包含字母 “X”，同时后面不跟有 “a” 或 “o” 的所有行：

```
grep 'X[^ao]' data
```

下述两条命令搜索所有包含至少一个非字母字符的行：

```
grep '^[^A-Za-z]' data
grep '^[^[:alpha:]]' data
```

提示

理解复杂正则表达式的技巧就是记住每个字符类——不管它看上去有多么复杂——只表示一个单独的字符。

20.7 区域设置和排序序列：locale；ASCII 码

此时，您可能想知道是否能使用其他范围替代预定义字符类。例如，能不能使用 **a-z**

替代[:lower:]? 同样, 能不能使用 A-Z 替代[:upper:]? 或者能不能使用 A-Za-z 替代[:alpha:]? 又或者使用 A-Za-z0-9 替代[:alnum:]?

答案是或许可以。在一些系统上这是可行的, 但是在其他系统上不可行。在解释为什么会是这种情况之前, 先讨论一下“区域设置”的思想。

当写 0-9 时, 它是 0123...9 的缩写, 这是确定的。但是, 当写 a-z 时, 它并不一定意味着 abcd...z。这是因为特定系统上的字母顺序依赖于所谓的“区域设置”, 而“区域设置”在各个系统之间可能有所不同。

为什么会是这种情况呢? 在 20 世纪 90 年代之前, Unix(及大多数计算机系统)使用的字符编码是 ASCII 码。名称 ASCII 代表 American Standard Code for Information Interchange, 即美国信息交换标准码。

ASCII 码创建于 1967 年。它为每个字符指定了一个 7 位的模式, 所以总共有 128 个字符。这些位模式的范围从 0000000(十进制 0)到 1111111(十进制 127)。ASCII 码包含第 7 章中讨论的所有控制字符, 以及 95 个可显示字符: 字母表的字母、数字和标点符号。可显示字符如下所示(注意第一个字符是空格)。

```
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

可显示字符的顺序就是上面列举的顺序。它们的范围从字符#32(空格)至字符#126(否定号)。出于参考目的, 附录 D 中示范了一个包含全部 ASCII 码的表(现在就可以去看一看)。

为了方便起见, 大多数 Unix 系统都有一个包含 ASCII 码的参考页。带来了很大的便利, 因为它允许快速方便地查看 ASCII 码。然而, ASCII 码页并不标准化的, 因此它的显示方式依赖于所使用的系统。详情请参见图 20-5。

Unix 类型	显示 ASCII 码页的命令
Linux	man ascii
FreeBSD	less /usr/share/misc/ascii
Solaris	less /usr/pub/ascii

图 20-5 显示 ASCII 码

本书附录 D 是一个 ASCII 码的汇总表。作为联机参考, 大多数 Unix 系统都有一个包含整个 ASCII 码的页面。传统上讲, 这个页面存储在目录/usr/pub 中的文件 ascii 中。最近几年, 一些系统重新组织了 Unix 文件系统, ASCII 参考文件被移至目录/usr/share/misc 中。在其他系统上, 该文件已经转换为联机手册中的页面。因此, 显示 ASCII 参考页的方式依赖于使用的系统。

正如第 19 章中讨论的, 在字符编码方案中, 字符组织的顺序称为排序序列。每当需要按顺序排放字符(例如排序数据或者在正则表达式中使用范围)时就需要使用排序序列。

当使用的是 Unix 或 Linux 时, 系统使用的排序序列依赖于使用的区域设置。区域设置的概念属于 POSIX 1003.2 标准, 我们已经在第 11 章和第 19 章中讨论过。正如第 19 章中解释的, 区域设置(由环境变量设置)告诉程序您希望使用哪种语言约定。这就使得世界上

的任何人都可以选择一种匹配自己母语的区域设置。

在一些 Unix 系统上, 区域设置被设置为其默认的排序序列与 ASCII 码中的字符顺序相匹配。从上面可以看出, 所有的大写字母都在一起, 而且位于小写字母的前面。这种序列称为 **C** 排序序列(参见第 19 章), 这是因为 C 编程语言使用该排序序列。

至于其他 Unix 系统, 包括许多 Linux 系统, 区域设置按如下方式设置, 即默认的排序序列将大写字母和小写字母成对组合: **AaBbCcDd...Zz**。这种排序序列的优点在于可以按照字典的顺序搜索单词或字符。因此, 这种排序序列被称为字典排序序列(参见第 19 章)。

对于正则表达式来说, 有时候可能会遇到问题, 因为 Unix 会根据系统使用的排序序列扩展 **a-z** 和 **A-Z**。

如果使用的是 **C** 排序序列, 那么所有的大写字母自成一组, 所有的小写字母也是如此。这意味着, 当指定所有的大写字母或小写字母时, 可以使用范围取代预定义字符类: **A-Z** 取代[:upper:], **a-z** 取代[:lower:], **A-Za-z** 取代[:alpha:]以及 **A-Za-z0-9** 取代[:alnum:], 如图 20-3 所示。

如果使用的是字典排序序列, 那么字母采用一种不同的顺序排列: **AaBbCcDd...Zz**。这意味着范围的工作方式不同了。例如, **a-z** 将表示 **aBbCcDd...YyZz**。注意这里没有大写字母 “A”(您知道原因吗?) 同样, **A-Z** 将表示 **AaBbCcDd...Z**, 没有小写字母 “z”。

作为示例, 假设您希望搜索文件 **data**, 查找所有包含从 “A” 到 “E” 的大写字母或小写字母的行。如果区域设置使用的是 **C** 排序序列, 则使用:

```
grep '[A-Ea-e]' data
```

在这个例子中, 该正则表达式等价于 **ABCDEabcde**, 而这也是大多数有经验的 Unix 用户所期望的。但是, 如果区域设置使用的是字典排序序列, 则必须使用:

```
grep '[A-e]' data
```

从传统上讲, Unix 过去使用的是 **C** 排序序列, 而许多老的 Unix 用户假定 **a-z** 总是指小写字母(唯一), **A-Z** 总是指大写字母(唯一)。但是, 这是一种不好的假定, 因为一些类型的 Unix, 包括许多 Linux 发行版, 都默认使用字典排序序列, 而不是 **C** 排序序列。但是, 不管系统默认使用哪一种排序序列, 有一种方式可以确保该排序序列是 **C** 排序序列, 而这是大多数 Unix 用户喜欢的。

首先, 需要确定系统默认使用了哪一种排序序列。为此, 先使用下述命令创建一个短文件 **data**:

```
cat > data
```

键入两行内容, 然后按[^]D 结束上述命令:

```
A
a
```

现在键入下述命令:

```
grep '[A-Z]' data
```


如果输出包含文件中的两行(A 和 a), 则使用的是字典排序序列。如果输出只包含含有 A 的这一行, 则使用的是 C 排序序列(为什么是这种情况呢?)

如果系统使用的是 C 排序序列, 则不需要做任何事情。但是, 请您阅读本节的剩余部分, 因为有一天, 您可能会在另一个系统上遇到该问题。

如果系统使用的是字典排序序列, 可以将它改变成 C 排序序列。为此, 需要将环境变量 **LC-COLLATE** 设置为 C 或 **POSIX**。对于 Bourne Shell 家族来说, 可以使用下述两条命令中的任意一条:

```
export LC_COLLATE=C
export LC_COLLATE=POSIX
```

对于 C-Shell 家族来说, 可以使用下述两条命令中的任意一条:

```
setenv LC_COLLATE C
setenv LC_COLLATE POSIX
```

要使改变永久化, 只需将其中一条命令放在登录文件中(登录文件在第 14 章讨论过, 环境变量在第 12 章中讨论)。

一旦确定系统使用的是 C 排序序列, 就可以用合适的范围替换预定义字符类, 如图 20-3 所示。本章的剩余内容将假定您使用的是 C 排序序列(因此, 如果您使用的不是 C 排序序列, 请立即在登录文件中添加上合适的命令)。

为了显示系统所使用区域设置的信息, 可以使用 **locale** 命令。**locale** 命令的语法为:

```
locale [ -a]
```

区域设置通过设置一组标准全局变量来维护, 其中包括 **LC_COLLATE**。为了查看这些变量的当前值, 可以输入命令本身:

```
locale
```

为了显示系统中所有可用的区域设置, 可以使用 **-a(all, 全部)** 选项:

```
locale -a
```

20.8 使用范围和预定义字符类

一旦确定使用的就是 C 排序序列(正如上一节中所述), 在指定模式时就拥有了一些灵活性。当希望匹配所有的大写字母或小写字母时, 既可以使用预定义字符类, 也可以使用范围。

例如, 下述两条命令都可以搜索文件 **data**, 查找所有包含字母 H, 后面跟任何从 a 到 z 的小写字母的行, 例如 “Ha”、“Hb”、“Hc” 等:

```
grep 'H[[:lower:]]' data
grep 'H[a-z]' data
```

接下来的两条命令搜索所有包含一个单独的大写字母或者小写字母，后面跟一个单独的数字，再后跟一个小写字母的行：

```
grep '[A-Za-z][0-9][a-z]' data
grep '[:alpha:][:digit:][:lower:]' data
```

下面是一个更复杂的例子——搜索加拿大邮政编码。加拿大邮政编码的格式为“字母 数字 字母 空格 数字 字母 数字”，其中所有的字母都是大写字母，例如 **M5P 3G4**。仔细地分析下面两条命令，直到完全理解它们：

```
grep '[A-Z][0-9][A-Z] [0-9][A-Z][0-9]' data
grep '[:upper:][:digit:][:upper:] [[:digit:][:upper:][:digit:]]' data
```

选择使用哪一种类型的字符类——是范围还是预定义名称——由自己决定。许多以前的 Unix 用户倾向于使用范围，因为他们学习的就是范围。此外，范围还比名称更容易键入，因为名称需要冒号和额外的方括号(参见前面的例子)。

但是，名称更可读，从而使得它们更适合在 shell 脚本中使用。另外，不管使用哪一种区域设置或语言，名称总是正确的，因此它们的移植性更出色。例如，假设处理的文本中包含非英语字符，如é(有重音符的 e)。通过使用[:lower:]，就可以确保获取é。但是，如果使用 a-z 的话，可能实现不了这种效果。

20.9 重复运算符

在正则表达式中，单个的字符(例如 A)或者字符类(例如 A-Za-z 或者[:alpha:])只匹配一个字符。为了一次匹配多个字符，可以使用**重复运算符**(repetition operator)。

最有用的重复运算符就是*(星号)元字符。一个*可以匹配前面字符的 0 次或多次出现。(有关“0 次或多次”的思想参见第 10 章中的讨论)。例如，假设您希望搜索文件 data，查找包含大写字母“H”，后面跟 0 个或多个小写字母的行。为此，可以使用下述两条命令中的任意一条：

```
grep 'H[a-z]*' data
grep 'H[[:lower:]]*' data
```

这些命令将查找类似于如下单词的模式：

```
H Har Harley Harmonica Harpoon HarDeeHarHar
```

最常见的组合就是使用一个.(点号)，后跟一个*。这将匹配任何字符的 0 次或多次出现。例如，下述命令搜索包含“error”，后跟 0 个或多个字符，再后跟“code”的行：

```
grep 'error.*code' data
```

作为示例，该命令将选取下面各行：

Be sure to document the error code.
 Don't make an error while you are writing code.
 Remember that errorcode #5 means "Too many parentheses".

下述例子搜索包含一个冒号, 后跟 0 个或多个其他任意字符, 再后跟另一个冒号的行:

```
grep ':.*:' data
```

有时候, 可能需要匹配 1 个或多个字符, 而不是 0 个或多个。为了这样做, 需要使用 +(加号)元字符替代*。例如, 下述命令搜索文件 **data**, 选取包含字符串 “variable”, 后跟 1 个或多个数字的行:

```
grep 'variable[0-9]+' data
grep 'variable[[:digit:]]+' data
```

这些命令将选取下面各行:

```
You can use variable1 if you want.
error in variable3x
address12, variable12 and number12
```

而不会选取下面各行:

```
Remember to use variable 1, 2 or 3.
variableX3 is the one to use
The next thing to do is set Variable417.
```

如果您希望在模式的开头同时匹配大写字母 “V” 和小写字母 “v”, 该怎么办呢? 只需将第一个字母改变成一个字符类即可:

```
grep '[vV]variable[0-9]+' data
```

下一个重复运算符是? (问号)元字符。这个重复运算符允许匹配某个实例 0 次或者 1 次。另一种说法就是? 使该实例是可选的。例如, 假设您希望查找文件 **data** 中所有包含单词 “color” (美国拼法)或 “colour” (英国拼法)的行, 则可以使用:

```
grep 'colou?r' data
```

最后一个重复运算符通过使用方括号创建所谓的限定(bound)来指定字符出现的次数。限定有 4 种不同的类型, 它们是:

```
{n}   正好匹配 n 次
{n,}  至少匹配 n 次
{,m}  最多匹配 m 次 [非标准]
{n,m} 至少匹配 n 次, 最多匹配 m 次
```

注意: 第三种结构 {,m} 并不属于 POSIX 1003.2 标准, 可能在一些程序中无法使用。

下面举一些例子。其中, 第一个例子正好匹配 3 个数字; 第二个例子至少匹配 3 个数字; 第三个例子最多匹配 5 个数字; 最后一个例子匹配 3 到 5 个数字。

```
[0-9]{3}
[0-9]{3,}
[0-9]{,5}
[0-9]{3,5}
```

为了示范如何在 **grep** 中使用一个限定，下述命令在文件 **data** 中查找所有包含 2 个数字或者 3 个数字的行。注意使用 **\<**和**\>**匹配整个数字：

```
grep '\<[0-9]{2,3}\>' data
```

到目前为止，我们只对单个的字符使用重复运算符。如果将多个字符用圆括号括起来，也可以对多个字符使用重复运算符。这样的模式称为**组**。通过创建组，可以将一串字符视为一个单元。例如，为了连续匹配字符串“xyz”5 次，可以使用下述两种正则表达式之一：

```
xyzxyzxyzxyzxyz
(xyz){5}
```

最后一个重复运算符是**|**(竖线)字符，它允许使用交变。也就是说可以匹配这一个，也可以匹配另一个。例如，假设我们希望在文件中搜索包含下述任意一个单词的行：

```
cat dog bird hamster
```

通过使用交变，搜索就简单了：

```
grep 'cat|dog|bird|hamster' data
```

很明显，这是一个功能强大的工具。但是，在这个例子中，您有没有发现一个问题？我们搜索的是字符串，而不是完整的单词。因此，上述命令同样也查找包含“concatenate”或“dogmatic”等单词的行。为了只查找完整的单词，需要明确匹配单词边界：

```
grep '\<(cat|dog|bird|hamster)\>' data
```

注意使用圆括号就是为了创建一个组。这将允许我们将整个模式视为一个单元。好好地想一想，直至明白其含义。

为了结束本节，我们解释最后一个元字符。众所周知，元字符在正则表达式中拥有特殊的含义。这就出现了一个问题：如果要匹配元字符，应该怎么办呢？例如，如果希望匹配一个真实的*(星号)、.(点号)或者| (竖线)字符，该怎么办呢？

答案就是使用****(反斜线)引用这些字符。这将把这些字符从元字符变成常规字符，从而可以从字面上解释这些字符。例如，为了搜索文件 **data**，查找所有包含“\$”字符的行，可以使用：

```
grep '\$' data
```

如果希望搜索反斜线本身，则只需连续使用两个反斜线。例如，为了查找所有包含字符“*”，后面跟任意数量的字符，再后跟 1 个或多个字母，最后跟一个“\$”的行，可以使用：

```
grep '\\\*.*[A-Za-z]+\$' data
```

20.10 理解复杂正则表达式的方式

一旦理解了正则表达式的规则，大多数正则表达式都比较容易编写。但是，正则表达式很难阅读，特别是当正则表达式比较冗长时。实际上，有经验的 Unix 人士在理解他们自己编写的正则表达式时也经常会遇到困难^{*}。下面提供一个简单的技巧，这是我数年来逐步总结的，可以帮助理解含义模糊的正则表达式。

当遇到难以理解的正则表达式时，将它写在一张纸上。然后将正则表达式分成不同部分，纵向书写不同部分，一个在另一个之上。依次取不同部分，并将各部分的含义写在同一行上。例如，考虑下述正则表达式：

```
\\*.*[A-Za-z]+\$
```

我们将这个正则表达式分隔成：

```

\\  → 1 个 \ (反斜线) 字符
\*  → 1 个 * (星号) 字符
.*  → 任意数量的其他字符
[A-Za-z]+ → 1 个或多个大写字母或小写字母
\$  → 1 个 $ (美元符号) 字符

```

当以这种方式分析正则表达式时，实际上是在以和程序处理该命令相似的方式解析该命令。经过一点练习之后，您会发现即便是复杂的正则表达式也变得可以理解了。

20.11 解决 3 个有趣的难题：字典文件

为了总结我们的讨论，我准备示范 3 个可以使用正则表达式解决的有趣难题。为了解决前两个难题，我们将使用一个文件，这个文件本身就很有趣，它就是字典文件。

字典文件从一开始就包含在 Unix 中，它包含有非常长的英语单词列表，其中包括大多数简明字典中经常使用的单词。由于每个单词一行，各行按字母顺序排列，因此该文件易于搜索。一旦习惯了使用字典文件，就能够完成各种令人惊异的事情。一些 Unix 命令，例如 **look** (第 19 章中讨论过)，就使用字典文件完成自己的工作。

字典文件的名称是 **words**。在早期版本的 Unix 中，**words** 文件存放在目录 **/usr/dict** 中。但是最近几年，Unix 文件结构进行了重组，在大多数现代系统上，包括 Linux 和 FreeBSD 在内，**words** 文件已被存放在目录 **/usr/share/dict** 中。在少数几个系统，例如 Solaris 上，这个文件存放在目录 **/usr/share/lib/dict** 中。因此，字典文件的路径名在各个系统之间可能有所不同(我们将在第 23 章中讨论 Unix 文件系统和路径名)。

^{*} 因此，正如一个谜语所言：“如果上帝什么都可以做，那么他能创建一个连他自己都不能理解的正则表达式吗？”

毫无疑问，这就是 Thomas Aquinas 在创作 *The Summa Theologica* 时所指的内容：“当我们说上帝可以做任何(all)事时，单词 ‘all’ 的准确含义将值得怀疑。”

出于参考目的，下面列出了查找字典文件的最常见位置：

```
/usr/share/dict/words
/usr/dict/words
/usr/share/lib/dict/words
```

在下面的例子中，我们将使用第一个路径名，因为这是最常见的。如果这个名称无效，可以试试另外两个。

我们从一个简单的问题开始。这个问题就是：哪些英语单词以“qu”开头并以“y”结尾呢？为了解决这个问题，只需使用下述正则表达式对字典文件进行 `grep`：

```
grep '^qu[a-z]+y$' /usr/share/dict/words
```

为了理解这个正则表达式，我们使用前面提到的技术。首先将这个正则表达式分隔成不同部分，并纵向书写不同部分，一个在另一个之上。正则表达式的分解如下所示：

```
^  → 行的开头
qu → 字符串“qu”
[a-z]+ → 1个或多个小写字母
y  → 字母“y”
$  → 行的结尾
```

记住字典文件中的每一行只包含一个单词。因此，我们从行的开头开始搜索，在行的结尾处结束搜索。

下一个难题是一个老问题。查找一个包含所有 5 个元音字母 a、e、i、o、u(并且以该顺序出现)的普通英语单词。这 5 个字母不必连在一起，但是它们必须按字母表顺序出现。也就是说，“a”必须位于“e”之前，而“e”必须位于“i”之前，等等。

为了解决这个问题，我们需要 `grep` 字典文件，查找包含字母“a”，后面跟 0 个或多个小写字母，再后跟一个字母“e”，再后跟 0 个或多个小写字母(后面的匹配过程依此类推)的单词。这一次，我们首先书写各个部分，然后再将各个部分组合在一起。当创建复杂的正则表达式时，这通常是一种有用的技术：

```
a  → 字母“a”
[a-z]* → 0个或多个小写字母
e  → 字母“e”
[a-z]* → 0个或多个小写字母
i  → 字母“i”
[a-z]* → 0个或多个小写字母
o  → 字母“o”
[a-z]* → 0个或多个小写字母
u  → 字母“u”
```

因此，完整的命令是：

* 正如第 19 章中所述，单词“grep”经常用作动词。


```
grep 'a[a-z]*e[a-z]*i[a-z]*o[a-z]*u' /usr/share/dict/words
```

为了避免不必要的悬念，我现在告诉您这条命令可以查找到许多单词，其中大多数单词都很偏僻。但是，有3个单词比较常见，它们是*：

```
adventitious
facetious
sacrilegious
```

最后一个问题就是在 Unix 文件系统中搜索历史产物。许多原始 Unix 命令的名称都是两个字母长：文本编辑器 **ed**，复制程序 **cp** 等。下面就查找所有这样的命令。

为了解决这个问题，我们需要知道最古老的 Unix 程序存放在 **/bin** 目录中。为了列举这个目录中的所有文件，可以使用 **ls** 命令(第24章讨论)：

```
ls /bin
```

为了分析 **ls** 的输出，可以将它的输出管道传送给 **grep**。当我们这样做时，**ls** 将自动地将每个名称放在一个单独的行上，因为输出将传送到过滤器。通过使用 **grep**，我们可以搜索只包含两个小写字母的行。完整的管道线如下所示：

```
ls /bin | grep '^[a-z]{2}$'
```

在一些系统上，**grep** 并不会返回预想的结果，因为它不能将方括号识别为元字符。如果出现这种情况，有两种选择。第一种是使用 **egrep** 替代 **grep**：

```
ls /bin | egrep '^[a-z]{2}$'
```

第二种就是消除方括号的需求，即简单地重复字符类，不再使用限定：

```
ls /bin | grep '^[a-z][a-z]$'
```

在自己的系统上试一试这些命令，看看会发现什么情况。当看到一个命令名称，希望查看该命令的更多信息时，可以在联机手册中查看该命令(参见第9章)。例如：

```
man ed cp
```

除了 **/bin** 中查找到的文件外，在 **/usr/bin** 中还有其他一些老 Unix 命令。为了在这个目录中搜索两个字符的命令名称，只需稍微修改一下前面的命令：

```
ls /usr/bin | grep '^[a-z]{2}$'
```

为了统计这样的命令有多少，可以使用带 **-c(count, 统计)** 选项的 **grep**：

```
ls /bin | grep -c '^[a-z]{2}$'
ls /usr/bin | grep -c '^[a-z]{2}$'
```

注意：当查看 **/usr/bin** 目录时，可能会发现一些两个字符的命令并不是老命令。为了

* 严格地讲，英语中有6个元音字母：a、e、i、o、u和(有时候)y。如果希望单词包含所有6个元音字母，只需将这3个单词转换为副词即可：“adventitiously”、“facetiously”和“sacrilegiously”。

查看命令是否来自初期的 Unix，可以查阅该命令的说明书页。

20.12 练习

1. 复习题

1. 什么是正则表达式？“正则表达式”的两个常见的缩写各是什么？
2. 在正则表达式中，解释下述元字符各匹配什么内容：`.`、`^`、`$`、`\<`、`\>`、`[list]`、`^[list]`。解释下述重复运算符各匹配什么内容：`*`、`+`、`?`、`{n}`。
3. 对于下面的每个预定义字符类，给出它们的定义并指定等价的范围：`[:lower:]`、`[:upper:]`、`[:alpha:]`、`[:digit:]`和`[:alnum:]`。例如，`[:lower:]`表示所有的小写字母，等价范围是 **a-z**。
4. 默认情况下，您的系统使用的是字典排序序列，但是您希望使用 C 排序序列。如何进行改变呢？对于 Bourne Shell 家族来说，应该使用什么命令呢？对于 C-Shell 家族来说，要使用什么命令呢？应该在哪个初始化文件中放置这样的命令呢？

2. 应用题

1. 创建匹配下述字符串的正则表达式并使用 `grep` 检测它们：

- “hello”
- 单词 “hello”
- 或者是单词 “hello”，或者是单词 “Hello”
- 行头的 “hello”
- 行尾的 “hello”
- 只包含 “hello” 的行

2. 使用重复运算符，创建匹配下述字符串的正则表达式：

- “start”，后面跟 0 个或多个数字，再后跟 “end”
- “start”，后面跟 1 个或多个数字，再后跟 “end”
- “start”，后面跟 0 个或 1 个数字，再后跟 “end”
- “start”，后面正好跟 3 个数字，再后跟 “end”

使用 `grep` 测试您的答案。提示：确保 `grep` 使用扩展(不是基本)正则表达式。

3. 正如我们在本章中讨论的，下述两条命令搜索文件 **data**，查找所有至少包含一个非字母字符的行：

```
grep '[^A-Za-z]' data
grep '[^[:alpha:]]' data
```

应该使用什么命令查找不包含任何字母字符的行？

4. 在 Usenet 全球讨论组系统中, 自由表达非常重要。但是, 重要的是人们应该尽量避免攻击性的帖子。解决方法就是对潜在的攻击性文本编码, 使其看起来就像无用信息一样。但是, 编码文本可以被某些人轻易地解码。

用来实现该编码的系统称为 Rot-13。它的工作原理如下所示。每个字母表中的字母都被字母表中该字母之后的第 13 个字母替换, 替换过程中, 在必要时返回到字母表的开头:

A → N	N → A
B → O	O → B
C → P	P → C
D → Q...	Q → D...

创建一条命令从文件 **input** 中读取数据, 使用 Rot-13 编码文本, 并将编码后的文本写入到标准输出。然后创建一条命令读取编码后的 Rot-13 数据, 并将它转换为普通文本。通过创建一个文本文件 **input** 测试您的解决方法, 编码并解码该文件。

3. 思考题

1. 术语“正则表达式”来源于抽象计算机科学概念。使用这样一个名称是一种好想法还是一种坏想法? 如果“正则表达式”这个术语被更直接的名称, 例如“模式匹配表达式”或者“模式匹配器”取代, 会有很大不同吗? 为什么?

2. 随着支持国际化的区域设置的引入, 已经使用了多年的正则表达式模式在一些系统上停止使用了。特别是依赖于传统 C 排序序列的正则表达式, 并不总是能与字典排序序列和平共处。大多数情况下, 解决方法就是使用预定义字符类替代范围。例如, 使用 **[!lower:]** 而不是 **a-z**(完整的集合请参见图 20-3)。您如何评价这种安排? 列举旧系统更出色的 3 个理由。列举新系统更出色的 3 个理由。



显示文件

在使用计算机的过程中，重要的是要提醒自己，我们努力的主要结果几乎总是获得某些类型的输出：文本、数字、图形、声音、图像、视频或一些其他数据。当使用本书讨论的 Unix 命令行程序时，输出通常是文本，在生成时或者显示在显示器上，或者保存到文件中。

基于这一原因，Unix 一直拥有许多程序可以用来显示文本数据，这些文本数据或者来自程序的输出，或者来自文件。在本章中，我们将讨论用来显示文件内容的程序。我们从文本文件的显示入手，然后再介绍二进制文件的显示。

在讨论过程中，我希望实现两个方面的目标。首先，无论何时，当需要显示文件中的数据时，应该能够分析场合，选择完成任务的最佳程序。其次，无论决定使用哪个程序，都应该对这个程序足够熟悉，从而能够处理大多数日常任务。

本章开始将首先综述用来显示文件的 Unix 程序。我们将介绍每个程序，解释它的用途并解释何时使用该程序。然后，依次讨论每个程序，这时候将阐述程序的细节。到目前为止，这类程序中最重要的是 `less` 程序(稍后再解释该名称的由来)。基于这一原因，我们将主要讨论这个最有用也最实用的程序。

在讨论各个程序时，我们还将讨论两个有趣的话题。第一个话题就是描述基于文本的程序处理输入的两种不同方式：“成熟模式(cooked mode)”和“原始模式(raw mode)”。第二个话题就是介绍二进制、八进制和十六进制的系统及概念，当显示二进制文件时必须理解这些概念。

尽管严格地讲，我还没有解释过文件实际上指什么，但是在本章中我们将讨论如何显示“文件”。在第 23 章中，将详细地讨论 Unix 文件系统。那个时候，我们将给 Unix 文件下一个准确的技术定义。现在，我们只做直观理解，即文件具有一个名称且包含一些信息。例如，您可以显示一个名为 `essay` 文件，该文件中包含您所写论文的文本。

在开始之前再解释一个观点：当谈论“显示”文件时，指的是显示文件的内容。例如，如果我写“下述命令显示文件 `essay`”，那么这意味着“下述命令显示文件 `essay` 的内容”。这是一个微妙，但是重要的观点，因此一定要理解它。

21.1 文件显示程序综述

Unix 中有许多显示文件的程序。在本节中，我们将综述这些程序，从而使您对有哪些程序可以使用有一个总体了解。在本章后面，我们将详细讨论每个程序。

首先，有些程序只是用来每次一屏地显示文本数据。这样的程序称为**分页程序**。该名称来源于下述事实，在 Unix 初期，用户只拥有将输出打印在纸上的终端。因此，为了查看文件，需要将文件每次一页地打印在纸上。当然，现在为了查看文件，需要每次一屏地在显示器上显示文件。然而，完成该作业的程序仍然称为“分页程序”。

通常，分页程序的使用有两种方式。第一种方式，正如第 15 章中讨论的，可以在管道线的末尾使用分页程序显示另一个程序的输出。在前面各章中，已经示范了许多这样的例子，例如：

```
cat newnames oldnames | grep Harley | sort | less
colrm 14 30 < students | less
```

在第一个管道线中，我们组合两个文件的内容，**grep** 所有包含字符串“Harley”的行，然后将结果发送给 **less** 进行显示。在第二个例子中，我们从一个文件读取数据，移除每行数据的第 14 列至第 30 列，然后将结果发送给 **less** 进行显示。

另一种使用分页程序的方式就是让分页程序每次一屏地显示一个文件的内容。例如，下述命令使用 **less** 查看 Unix 口令文件(第 11 章描述过)的内容：

```
less /etc/passwd
```

通过这种方式可以查看任何文本文件，只需键入 **less**，后面跟着文件的名称即可(本章后面将讨论 **less** 程序的选项、语法和其他细节)。

尽管 **less** 是最主要的 Unix 分页程序，但是还有其他两个大家也可能听说过的同类程序：**more** 和 **pg**。在第 2 章的讨论中讲过，20 世纪 80 年代，Unix 有两个主要分支：AT&T 公司开发的 System V 和加利福尼亚大学伯克利分校开发的 BSD。**pg** 程序是 System V 的默认分页程序，而 **more** 是 BSD 的默认分页程序。现在，这两种程序都已经废弃，被 **less** 所取代。

在一些罕见场合中，有时候不得不使用 **more**。基于这一原因，我们将稍微讨论一下 **more**，以便在您遇到它时，知道如何使用它。在大多数情况下，**pg** 程序已经消逝和抛弃，因此没有必要再讨论这个程序。这里提及它主要是由于历史原因：如果您看到这个名称，至少应该知道它是干什么用的。

除了使用分页程序外，还可以使用 **cat** 程序显示文件。正如第 16 章中讨论的，**cat** 的主要应用是组合多个文件的内容。但是，**cat** 还可以用于快速地显示文件，例如：

```
cat /etc/passwd
```

因为 **cat** 程序一次显示整个文件(不是每次一屏)，所以只有文件足够短小，屏幕不需要滚动就可以完全显示时，才可以使用它。大多数时候，最好还是使用 **less**。

大多数情况下，当希望查看整个文件时，可以使用 **less** 或者 **cat**。如果只希望显示文

件的一部分, 则还有其他 3 个程序可供使用: 显示文件开头的 **head**、显示文件末尾的 **tail** 以及显示所有包含(或不包含)特定模式的行的 **grep**。

在第 16 章中, 我们讨论了如何在管道线中将 **head** 和 **tail** 作为过滤器使用。在本章中, 将示范如何在文件中使用它们。在第 19 章中, 我们详细地讨论了 **grep**, 而且在第 20 章也示范了许多例子。基于这一原因, 我们不需要在本章中再讨论 **grep**(但是, 我确实希望提及它)。

接下来可以用来显示文件的一组程序就是文本编辑器。文本编辑器允许执行查看文件的任何部分、搜索模式、在文件中向后及向前移动等动作。它还允许编辑(改变)文件。因此, 当需要在显示文件的同时改变文件, 或者希望使用特殊的编辑器命令在文件中移动时, 可以使用文本编辑器显示文件。否则, 应该使用分页程序。

在第 14 章中, 我们已经提及了几种在 Unix 系统和 Linux 系统中广泛使用的文本编辑器: **kedit**、**gedit**、**Pico**、**Nano**、**vi** 和 **Emacs**。这些编辑器中的任何一个都允许显示和修改文件。其中, **vi** 和 **Emacs** 是目前为止功能最强大的工具(并且是最难学习的工具)。本书中将详细讨论的唯一编辑器是 **vi**, 其细节信息参见第 22 章。

有时候, 可能希望使用文本编辑器查看某个非常重要的文件, 但是不希望由于不小心而修改了这个文件。在这种情况下, 可以以所谓的“只读”模式运行编辑器, 这意味着只能查看文件, 但是不能进行任何改变。

为了以只读模式启动 **vi**, 可以使用 **-R** 选项。例如, 任何用户都可以查看 Unix 口令文件(参见第 11 章), 但是除了超级用户, 其他用户不允许修改它。因此, 为了使用 **vi** 文本编辑器查看口令文件, 但又不允许编辑它, 可以使用:

```
vi -R /etc/passwd
```

为了方便起见, 可以将 **view** 作为 **vi -R** 的同义词使用:

```
view /etc/passwd
```

即便以超级用户登录, 您也应该经常选择使用 **vi -R** 或者 **view** 查看非常重要的系统文件。这样可以确保不会由于不小心而改变文件(我们将在第 22 章中详细讨论这一点)。

到目前为止, 我们讨论的程序针对的都是文本文件, 也就是包含字符行的文件。但是, 还有许多不同类型的非文本文件, 我们称之为二进制文件。有时候, 也可能需要查看这样的文件的内部。我希望提及的最后两个程序是 **hexdump** 和 **od**, 它们用来显示包含二进制数据的文件。

例如, 假设您在编写一个将二进制输出发送给文件的程序。每次运行该程序时, 您需要查看这个文件的内部, 以检查输出。这就是 **hexdump** 或 **od** 程序发挥作用的地方。我们将在本章后面详细地讨论它们。作为一个快速的例子, 下述两条命令都可以查看包含 **grep** 程序的文件的内部(现在还不用关心选项, 我们稍后再讨论它们)。

```
hexdump -C /bin/grep | less  
od -Ax -tx1z /bin/grep | less
```

出于参考目的, 图 12-1 包含了本综述中讨论的各个程序的摘要信息。在查看该摘要信

息时, 请注意共有多少个 Unix 工具可以用来显示文件, 每个工具都有其各自的特征和用处。

程序	作用	章号
less	分页程序: 每次一屏地显示数据	21
more	分页程序(已经废弃, 在 BSD 中使用)	21
pg	分页程序(已经废弃, 在 System V 中使用)	—
cat	显示整个文件, 没有分页	16
head	显示文件的第一部分	16、21
tail	显示文件的最后一部分	16、21
grep	显示包含/不包含特定模式的行	19、20
vi	文本编辑器: 显示和编辑文件	21
view、vi -R	只读文本编辑器: 显示但不允许修改文件	22
hexdump	显示二进制(非文本)文件	21
od	显示二进制(非文本)文件	21

图 21-1 文件显示程序

Unix 系统和 Linux 系统拥有大量可以用来显示文件全部或部分内容的工具。本摘要示范了其中最重要的几种工具, 同时还给出了讨论它们的章号。至少, 您应该能够使用 **less**、**cat**、**head**、**tail** 和 **grep** 显示文本文件。另外, 还应该知道如何使用 **vi**, 因为它是最主要的 Unix 文本编辑器。如果您是一名程序员, 那么您还应该熟悉 **hexdump** 或者 **od**, 以便能够显示二进制文件。

21.2 less 简介: 启动、停止、帮助

less 程序是一个分页程序。也就是说, 它每次一屏地显示数据。当启动 **less** 时, 可以选择使用许多选项, 而一旦 **less** 运行, 还有许多命令可供使用。但是, 极少情况下才需要如此复杂。在本章中, 我们只关注那些在日常应用中可能使用的基本选项和特性。本章并不提供更深奥的选项和命令的描述, 这些内容可以参见说明书页和 Info 页:

```
man less
info less
```

(联机手册和 Info 系统在第 9 章中讨论过。)

使用 **less** 的基本语法如下所示:

```
less [-cCEfMmSX] [+command] [-xtab] [file...]
```

其中 *command* 是 **less** 自动执行的一条命令, *tab* 是希望使用的制表间距, *file* 是文件的名称。

大多数时候, 不需要任何选项。所需做的全部工作就是指定一个或多个要显示的文件, 例如:

```
less information
less names addresses
```

您可以使用 **less** 显示任何自己有权限阅读的文本文件的内容, 包括系统文件或者属于另一个用户标识的文件(我们将在第 25 章中讨论文件权限)。作为示例, 下述命令显示一个著名的系统文件, 即第 7 章中讨论的 **Termcap** 文件:

```
less /etc/termcap
```

Termcap 文件中包含所有不同类型的终端的技术描述。尽管 **Termcap** 在很大程度上已经被一个称为 **Terminfo** 的新系统(参见第 7 章)所替代*, 但是这个文件是一个练习使用 **less** 的极好例子, 因此, 如果希望遵循本章的描述流程, 可以在任何时候输入上述命令。

在显示任何内容之前, **less** 将清除屏幕(可以使用 **-X** 选项禁止该操作)。当 **less** 启动时, 它显示第一屏数据, 这屏数据正好填满显示器或者窗口。在屏幕的左下角会看到一个提示。初始提示包含被显示文件的名称。根据系统的配置, 可能还会看到其他信息。例如:

```
/etc/termcap lines 1-33/18956 0%
```

在这个例子中, 我们查看的是文件 **/etc/termcap** 的第 1 行至第 33 行。屏幕最顶端的一行就是文件的第一行(0%)。随后的提示将更新行号和百分比。

在一些系统上, 默认设置是 **less** 显示较简单的提示, 没有行号和百分比。如果系统是这种情况, 那么第一个提示将只显示文件名, 例如:

```
/etc/termcap
```

随后的提示将更加简单, 只是一个冒号:

```
:
```

在这样的系统上, 可以使用 **-M** 选项在提示中显示额外的信息(本章后面讨论)。

提示

对于有大量额外时间的爱好者而言, **less** 在提示定制方面, 要比历史上的其他分页程序拥有更多的灵活性(细节请参见说明书页)。

一旦看到提示, 就可以输入命令。一会之后, 我们将讨论各种命令。这一类命令有许多, 现在, 我们只提及最常见的命令, 即简单地按 **<Space>** 键。这将告诉 **less** 显示下一屏数据。这样就可以每次一屏, 从头到尾地按 **<Space>** 键阅读整个文件。

当到达文件的末尾时, **less** 将提示改变为:

```
(END)
```

如果希望退出 **less**, 可以在任何时候按 **q** 键。这样可以不必等到达文件的末尾再退出 **less**。因此, 为了查看文件, 只需启动 **less**, 一直按 **<Space>** 键, 直到看到希望的内容, 然

* 尽管 **Terminfo** 系统是首选(参见第 7 章), 但是一些程序仍然使用 **Termcap**, 包括 **less** 本身。

后按 **q** 键退出。

作为一个快速的练习，试试下面的例子。输入下述命令之一显示 `termcap` 文件：

```
less /etc/termcap
less -m /etc/termcap
```

您将看到第一屏数据。按 `<Space>` 键几次，每次一屏地向文件后面移动。在查看无边无际、深奥、过时的终端描述的过程中，当感觉疲惫时，可以按 **q** 键退出。

提示

当使用 `less` 显示文件时，有许多命令可以在查看文件的过程中使用。其中最重要的命令就是 **h**(help, 帮助)。在任何时候您都可以按 **h** 键显示一个所有命令的列表。

学习 `less` 的最佳方式就是按 **h** 键，查看有什么可用的命令并进行体验。

21.3 less 和 more 的故事

正如本章前面所解释的，原始的 Unix 分页程序是 `more`(在 BSD 中使用)和 `pg`(在 System V 中使用)。有时候，您可能听到名称 `less` 来自于一个扭曲的笑话。因为 `less` 要远比 `more` 的功能更为强大，所以这个笑话就是“`less is more`(`less` 就是更强大的)”。这看起来有道理，但事实并非如此。下面才是真实的故事。

原始的 Unix 分页程序 `more` 是一个简单的程序，用来每次一屏地显示数据。名称 `more` 来自下述事实，即在每屏之后，程序都显示一个提示，这个提示中有一个单词“**More**”：

```
--More--
```

`more` 程序非常有用，但是它有严重的局限性。其中最重要的局限性就是 `more` 只能从头到尾地显示数据，而不能倒退。

1983 年，程序员 Mark Nudelman 在 Integrated Office Systems 公司工作。该公司生产能够创建非常大型的包含事务信息和错误消息的日志文件的 Unix 软件。有些文件非常大，以至于当时版本的 `vi` 文本编辑器不能够读取它们。因此，Nudelman 和其他程序员在希望查看错误时，被迫使用 `more` 查看文件。

但是，这里有一个问题。每当程序员在日志文件中发现一个错误消息时，没有办法退回去查看是什么情况导致了这一问题，也就是查看直接位于错误前面的事务信息。程序员经常抱怨这个问题。Nudelman 曾经对我解释过：

“一组工程师围在实验室中的终端周围，使用 `more` 查看日志文件。我们发现有一行内容显示发生了错误，和往常一样，我们不得不确定错误的行号。然后，不得不退出 `more`，重新启动它，并向前移动，移动到发生错误的那一行代码的前几行，查看是什么原因导致了错误。一些人抱怨这种麻烦的过程。有人说：‘我们需要一个向后退的 `more`’。另一个人说：‘是，我们需要 `LESS`!’，这使得每个人都发出了笑声。”

经过考虑, Nudelman 认为创建一个能够后退的分页程序并不太难。在 1983 年末, 他编写了这样一个程序, 并且真的称这个程序为 **less**。刚开始时, **less** 只在公司内部使用。但是, 在增强程序之后, Nudelman 认为可以在外部公开发行, 所以他在 1985 年 5 月将它公开。

Nudelman 以开放源代码软件的方式发布了 **less**, 从而允许许多其他人帮助他改进程序。几年以后, **less** 功能越来越强大, 在 Unix 用户中流行起来, 并最终取代了 **more** 和 **pg**(其他流行的 Unix 分页程序)。现在, **less** 是世界上使用最广泛的 Unix 分页程序, 并作为 GNU 实用工具(参见第 2 章)发行。

有趣注释: 大多数程序发行时使用 1.0、1.01、1.2 和 2.0 等形式的版本号。Nudelman 使用了一种比较简单的版本系统。从一开始, 他就将每个新版本的 **less** 按自己的数字命名: 1、2、3、4 等。因此, 在编写这部分内容时, 我使用的 **less** 程序的版本号为 394。

21.4 使用 less

在使用 **less** 阅读文件时, 有许多命令可以使用。出于参考目的, 图 21-2 列举了最重要的命令。要查阅更综合的命令一览表, 可以在 **less** 程序内按 **h**(help, 帮助)键, 或者在 shell 提示处输入下述命令:

```
less --help
```

当显示这个命令一览表时, 将会看到许多从来没有使用过的命令。例如, 向前移动(也即向下移动)一行就有 5 种不同的方式, 向后(上)移动一行也有 5 种不同的方式, 退出程序也有 5 种不同的方式。不要害怕, 其实您只需知道图 21-2 中列举的命令。

最好的策略就是首先从前面提到的 3 条命令入手: 在文件中向前移动一屏的 **<Space>**、请求帮助的 **h** 以及退出程序的 **q**。一旦熟悉了这 3 条命令, 就可以自学图 21-2 中所示的其余命令, 每次一个, 直到全部记住它们。在学习过程中, 可以按照从顶到底的顺序学习(我精心安排了这些命令的顺序, 而且这些命令确实需要记住)。

如果需要一个练习文件, 则可以使用本章前面提到的 **termcap** 文件。下述命令将使您踏入学习之旅:

```
less -m /etc/termcap
```

在练习过程中使用 **-m** 选项会对您有所帮助, 使用这个选项可以使提示显示您在文件中的位置。

基本命令	
h	显示帮助信息
<Space>	前进一屏
q	退出程序
高级命令	
g	跳到第一行
G	跳到最后一行
=	显示当前行号和文件名
<Return>	前进一行
n<Return>	前进 <i>n</i> 行
b	后退一屏
y	后退一行
ny	后退 <i>n</i> 行
d	前进(向下)半屏
u	后退(向上)半屏
<Down>	前进一行
<Up>	后退一行
<PageUp>	后退(向上)一屏
<PageDown>	前进(向下)一屏
ng	跳到第 <i>n</i> 行
np	跳到文本的 <i>n</i> %行处
/pattern	向前搜索指定模式
?pattern	向后搜索指定模式
n	重复搜索: 相同方向
N	重复搜索: 相反方向
!command	执行指定的 shell 命令
v	使用当前文件启动 vi 编辑器
-option	改变指定的 <i>option</i>
_option	显示 <i>option</i> 的当前值

图 21-2 less: 最有用命令一览表

21.5 使用 less 在文件中搜索

图 21-2 中的大多数命令都比较简单。但是, 我依然希望讨论一下搜索命令。当希望搜索某种模式时, 可以使用 **/**(向前搜索)或 **?**(向后搜索), 后面跟一个模式。模式可以是一个

简单的字符串或者正则表达式(第 20 章描述过)。在键入/或者?，后面跟一个模式之后，需要按<Return>键让 **less** 知道已经结束键入。

下面举一些例子。为了在文件中向前搜索“buffer”的下一出现，可以使用：

```
/buffer
```

为了向后搜索同一模式，可以使用：

```
?buffer
```

搜索是区分大小写的，因此如果搜索“Buffer”将会得到不同的结果：

```
/Buffer
```

如果希望使用不区分大小写的搜索，可以使用-I 选项启动 **less**(本章后面描述)，例如：

```
less -Im /etc/termcap
```

当以这种方式启动 **less** 时，对“buffer”的搜索将和对“Buffer”或“BUFFER”的搜索产生相同的结果。

如果希望在阅读文件的过程中将选项-I 关闭或打开，可以在 **less** 程序中使用-I 命令。为了显示该选项的当前状态，可以使用_I 命令(有关在 **less** 中改变和显示选项的内容在本章后面描述)。

如果希望执行其他类型的搜索，可以使用正则表达式。例如，假设您希望搜索任何包含“buf”，后面跟 0 个或多个小写字母的字符串，则可以使用：

```
/buf[:lower:]*
```

```
?buf[:lower:]*
```

有关正则表达式以及众多示例的详细解释，请参见第 20 章。

一旦输入了搜索命令，就可以使用 **n(next)** 命令重复执行它。这将以相同的方向执行完全相同的搜索。为了以相反的方向执行相同的搜索，需要使用 **N**。

每当搜索模式时，**less** 将高亮显示文件中出现的匹配模式。因此，在搜索某些东西时，在按页浏览文件的过程中，可以方便地看到所有匹配的结果。高亮显示将一直保持，直到输入另一个搜索。

提示

一旦学习了如何使用 **vi** 编辑器，**less** 命令将更容易理解，因为有许多命令直接取自 **vi**。这是因为几乎所有有经验的 Unix 人士都熟悉 **vi**，因此使用相同的命令会使 **less** 更容易理解。

提示

如果您有足够的空闲时间，可以尝试使用 **lesskey** 命令改变 **less** 所使用的键。有关细节请参见 **lesskey** 的说明书页。

21.6 原始模式和成熟模式

在继续讨论之前,我希望先介绍几个重要的 I/O(输入/输出)概念,这些概念可以帮助更好地理解 **less** 以及相似程序的工作原理。首先,我们介绍一下定义。

设备驱动程序(或者更简单一点,称为**驱动程序**)就是一个为操作系统和特定类型的设备(通常是某些类型的硬件)提供接口的程序。当使用 Unix 的基于文本的 CLI(命令行界面)时,控制终端的驱动程序称为**终端驱动程序**。

与其他一些驱动程序不同,终端驱动程序必须提供一个交互式用户界面,这就要求对数据进行特殊的预处理和后处理。为了满足该需求,终端驱动程序使用了所谓的**线路规程**(line discipline)。

Unix 有两种主要的线路规程:**规范模式**(canonical mode)和**原始模式**(raw mode)。线路规程的技术定义十分晦涩,但是其基本思想是,在规范模式中,键入的字符累积在一个缓冲区(存储区域)中,除非按下<Return>键,否则不会向程序发送任何东西。在原始模式(也称为**非规范模式**)中,只要键一按下,字符就直接发送给程序。当阅读 Unix 文档资料时,通常会看到将规范模式称为**成熟模式**(cooked mode)。当然,这是一个有趣的隐喻,因为“cooked,成熟”是“raw,原始”的反义词。

当程序员创建程序时,他可以使用任意一个线路规程。原始模式允许程序员完全控制用户的工作环境。例如,**less** 程序就工作在原始模式中,因此它能够完全接管命令行和屏幕,根据自己的需要显示行及处理字符。

这就是为什么无论何时,当按下键时,**less** 能够立即响应的原因。它不需要您按<Return>键。因此,只需简单地按<Space>键,**less** 就可以显示更多的数据;按 **b** 键就可以使 **less** 在文件中后退一屏;按 **q** 键就可以使程序退出。还有许多其他程序也采用原始模式,例如文本编辑器 **vi** 和 **Emacs**。

在规范(成熟)模式中,程序发送整行,而不是单个字符。这将使程序员免除在每个字符生成时都不得不对其进行处理的麻烦。另外,它还允许在行处理之前对行进行修改。例如,可以在按<Return>键之前使用<Backspace>或者<Delete>键进行修正。例如,当使用 **shell** 时,所使用的就是规范模式:除非按下<Return>键,否则不发送任何东西。

几乎所有基于文本的交互式程序都或者使用规范模式,或者使用原始模式。但是,您还有可能听说第三种线路规程,尽管它已经不经常使用。

cbreak 模式是原始模式的变种。大多数输入都直接发送给程序,就像原始模式一样。但是,有少数几个非常重要的键是由终端驱动程序直接处理的。这些键(在第 7 章中讨论过)就是发送 5 个特殊信号的键:**intr(^C)**、**quit(^\\)**、**susp(^Z)**、**stop(^S)**和 **start(^Q)**。因此,**cbreak** 模式主要是原始的,但也有一点成熟。过去,它还有个古怪的名称“稀有模式”。

21.7 less 使用的选项

当启动 **less** 时, 有大量的选项可供使用, 其中大多数选项都可以忽略^{*}。实际上, 可以认为 **less** 拥有下述语法:

```
less [-cCEFMms] [+command] [-xtab] [file...]
```

其中 *command* 是 **less** 自动执行的命令, *tab* 是希望使用的制表间距, 而 *file* 则代表文件的名称。

3 个最有用的选项是 **-s**、**-c** 和 **-m**。其中 **-s**(squeeze, 挤压)选项将多个空白行替换为一个空白行。这在压缩输出方面非常有用, 因为多个空白行并没有实际意义。当然, 原始文件并没有变化。

-c(clear, 清除)选项告诉 **less** 从顶端向下显示每一屏数据。如果没有 **-c** 选项, 新行将从屏幕底部向上滚动。与此相比, 使用 **-c** 选项可以方便长文件的阅读。**-C**(大写字母 “C”)选项与 **-c** 选项相似, 只是 **-C** 选项在写新数据之前将整个屏幕清空。最好您自己试试这两个选项, 看看喜欢哪一个。

名称 **-m** 指的是 **more**, 即前面提到的原始的 BSD 分页程序。**more** 提示显示一个百分比, 告诉用户已经 “浏览” 了文件的多大部分。在开发 **less** 时, 开发人员使用了一个简单的提示, 即一个冒号。但是, 为了方便那些已经习惯使用 **more**, 并且希望更多详细提示的用户, **less** 也提供了 **-m** 选项。

-m 选项通过展示已显示内容占文件的百分比, 使 **less** 的提示看上去更像 **more** 的提示。例如, 假设使用 **-m** 选项显示 **termcap** 文件(参见本章前面的讨论):

```
less -m /etc/termcap
```

在向下移动了一段距离之后, 将看到类似于下面的提示:

```
40%
```

这表示现在您已经 “浏览” 了 40% 的文件内容(顺便说一下, 可以使用命令 **40p** 直接跳到这个位置上来, 参见图 21-2)。

-M(大写字母 “M”)选项使提示显示更多的信息: 文件名和行号, 以及文件已经显示的内容所占的百分比。例如, 如果使用:

```
less -M /etc/termcap
```

那么典型的提示类似于下述内容:

```
/etc/termcap lines 7532-7572/18956 40%
```

行号指所显示各行的范围, 在这个例子中, 就是第 7532 行到第 7572 行(总共 18956 行)。

-E(end, 结尾)是我最喜欢的选项之一。它告诉 **less** 当显示到文件的末尾时, 自动退出

^{*} 实际上, **less** 就是一条古怪的命令, 与 **ls**(参见第 24 章)相似, **less** 拥有的选项比字母表中的字母还多。具体原因很难解释, 但是我怀疑这可能与甲状腺功能亢进有关。

程序。使用**-E**选项时，不用按 **q** 键就可以退出程序。当只需要浏览一遍文件，而不需要后退时，这个选项非常方便。

-F(finish automatically, 自动结束)告诉 **less** 在整个文件可以一次显示时自动退出程序。该选项也能够不用按 **q** 键就退出程序。依我的经验来看，**-F** 适合于非常短的文件，而**-E** 适合于长文件。为了明白这一点，让我们创建一个非常短的文件 **friends**。首先，输入命令：

```
cat > friends
```

现在键入五到六个朋友的姓名，每行一个。当完成时，按[^]**D** 发送 **eof** 信号结束程序([^]**D** 在第7章讨论过)。现在，比较下述两条命令。注意第二条命令，使用该命令时不用按 **q** 键就可以退出程序。

```
less friends
less -F friends
```

+(加号)选项允许指定 **less** 从什么地方开始显示数据。**+**号之后出现的任何内容都会作为初始命令执行。例如，为了以 **termcap** 文件的末尾作为初始位置显示该文件，可以使用：

```
less +G /etc/termcap
```

为了显示相同的文件，但是从搜索到的单词“**buffer**”开始，可以使用：

```
less +/buffer /etc/termcap
```

为了从一个特定的行开始，可以使用⁺**g**(go to, 跳到)选项，前面加上希望跳到的行号。例如，从第37行开始显示，可以使用：

```
less +37g /etc/termcap
```

为了方便起见，**less** 允许省略 **g**。因此，下述两条命令都从第37行开始显示：

```
less +37g /etc/termcap
less +37 /etc/termcap
```

-I(ignore case, 忽略大小写)选项告诉 **less** 在搜索模式时，忽略大写字母和小写字母之间的区别。默认情况下，**less** 是区分大小写的。例如，“**the**”就和“**The**”不同。但是，当使用了**-I**选项后，搜索“**the**”、“**The**”或“**THE**”都将获得相同的结果。

-N(number, 数字)选项用于在输出中显示行号。当使用这个选项时，**less** 为每行编号，就像 **nl** 命令(参见第18章)一样。例如，下面两个例子生成相似的输出：

```
less -N file
nl file | less
```

当然，在两个例子中，实际文件并没有改变。

nl 和 **less -N** 之间有两点重要的区别。首先，**less** 的行号只有一种形式，即1、2、3等。而 **nl** 命令拥有许多选项，在生成行号时有极大的灵活性。**nl** 命令可以选择起始号、增量等(参见第18章)。其次，**less** 对所有行都编号，即使是空白行。而默认情况下，**nl** 不对空白

行编号，除非使用**-b a** 选项。

最后，**-x** 选项后跟一个数字告诉 **less** 按指定的正则区间(regular interval)设置制表符。这将控制包含制表符字符的数据的间距。例如，为了显示一个制表符设置为 4 个空格的程序文件 **foo.c**，可以使用：

```
less -x4 foo.c
```

对于大多数 Unix 程序来说，默认的制表符设置是 8 个空格(参见第 18 章)。

在查看文件的过程中，如果希望改变一个正在使用的选项，可以使用**-(连字符)**命令，后面跟新的选项。这就像一个切换开关。例如，为了在查看文件的过程中打开**-M** 选项(显示详细的提示)，可以键入：

```
-M
```

为了关闭该选项，只需再次输入这条命令。

为了显示选项的当前值，可以使用**_**(下划线)，后面跟着选项。例如，为了查看提示是如何设置的，可以使用：

```
_M
```

下面再举一个例子。您已经在没有使用**-I** 选项的情况下启动了 **less**，正在查看文件，现在您决定执行不区分大小写的搜索。那么您所需做的就是键入：

```
-I
```

在输入搜索命令之后，再次键入**-I** 将关闭该选项。像这样来回几次之后，很可能不知道选项是打开的还是关闭的。所以，在任何时候，都可以键入下述命令查看该选项的状态：

```
_I
```

这是一个很方便的模式，需要记住。

提示

当您还不熟悉 **less**，但是又想学习如何使用各种选项时，您可以在显示文件的过程中使用**-(改变选项)**和**_**(显示选项)命令进行体验。

如果您希望学习如何使用**-P** 选项(本书没有讨论)改变提示，那么这种方式特别有用。使用这种方式改变提示后，立即就可以看到结果。

21.8 使用 less 和 cat 的时机

正如本章前面所讨论的，**less** 和 **cat** 都可以显示文件。对于 **less** 来说，文件每次只显示一屏；而对于 **cat** 来说，一次就显示整个文件。如果文件的长度比屏幕的大小要大，那么最好使用 **less**。如果使用 **cat** 的话，那么大部分文件内容在阅读之前就滚动出屏幕的范围。但是，如果是小文件呢？

如果使用 **cat** 显示小文件——小得足以在屏幕上显示，那么数据将快速地显示，而这正是 **cat** 的特点。这种情况下 **less** 就逊色很多，原因有两点。首先，**less** 将清空屏幕，消除以前的输出。其次，要退出程序时不得不按 **q** 键，当希望快速地显示几行内容时这多少有点烦人。

当然，也可以对 **less** 程序使用 **-F**(finish automatically, 自动结束)选项，从而使其在整个文件可以一次显示时能够自动退出程序。例如，假设 **data** 是一个非常小的文件。使用下述命令可以快速地显示这个文件：

```
less -F data
```

实际上，甚至可以指定 **less** 默认使用 **-F** 选项(通过设置 **LESS** 环境变量来指定，本章后面解释)。一旦设置了这个变量，就不必再键入 **-F** 了，下述两条命令基本上等价(假定 **data** 是一个非常小的文件)：

```
less data  
cat data
```

但是，如果您观察有经验的 Unix 人士，就会发现他们总是使用 **cat** 显示短文件。他们从不使用 **less**，这是为什么呢？

原因有 4 个方面。首先，正如前面所述，**less** 清空屏幕，删除前面的输出。这可能会产生不便。其次，“cat”的键入要比“less”的键入快。第三，名称 **cat** 要比名称 **less** 更可爱。最后，以这种方式使用 **cat** 就是 Unix 人士与众不同的原因所在。

这些看上去并不是什么重要的原因，但是 Unix 人士喜欢他们的工作平稳、快速而且快乐。因此，如果希望自己看上去像一个真正的 Unix 人士，而不是一个庸俗的人，请在文件非常小时使用 **cat**，否则使用 **less**。

21.9 使用环境变量定制分页程序

正如第 15 章中讨论的，Unix 的设计准则假设每个工具应该只做一件事情，且出色地完成这件事情。因此，Unix 分页程序(**less**、**more**、**pg**)都被设计为仅提供一种服务：每次一屏地显示数据。如果另一个程序需要该功能，那么这个程序不必自己提供，而是可以使用分页程序。

最常见的例子就是使用 **man** 程序(参见第 9 章)访问 Unix 联机手册。**man** 程序实际上并不显示页面的文本，而是调用一个分页程序，每次一屏地显示页面。

这就出现了问题，**man** 和其他程序使用哪个分页程序呢？您可能会认为，因为 **less** 是最流行的分页程序，所以任何需要这种工具的程序都会自动地使用 **less**。通常情况下是这样，但并不绝对。例如，在一些系统上，默认情况下 **man** 程序使用 **more** 显示说明书页。这可能有点不合情理，因为根据前面的讨论，**more** 没有 **less** 的功能强大。

但是，默认分页程序可以指定，即设置环境变量 **PAGER**，使其指向希望使用的分页程序。例如，下述命令将 **less** 设置为默认的分页程序。第一条命令是针对 Bourne Shell 家

族(Bash、Korn Shell)的。第二条命令是针对 C-Shell 家族(C-Shell、Tcsh)的。

```
export PAGER=less
setenv PAGER less
```

为了使改变永久化,可以将合适的命令放在登录文件中(环境变量在第 12 章中讨论过,登录文件在第 14 章中讨论过)。

一旦以这种方式设置了 **PAGER** 环境变量,所有需要外部分页程序的程序都将使用 **less**。即使 **less** 已经是系统首选的分页程序,最好也还是在登录文件中设置 **PAGER**。这将明确覆盖其他默认值,确保不管系统如何设置都可以让 **less** 成为首选。

除了 **PAGER**,还有一个环境变量可以用来做进一步的定制。这个环境变量就是 **LESS**,这个变量用来设置程序每次启动时希望使用的选项。例如,假设您希望在 **less** 中总是使用 **-CFMs** 选项(本章前面讨论过)。下述命令将设置 **LESS** 变量(第一条命令针对的是 Bourne Shell 家族,第二条命令针对的是 C-Shell 家族)。

```
export LESS='-CFMs'
setenv LESS '-CFMs'
```

这两条命令是放在登录文件中的命令。一旦这样做了, **less** 将默认以这些特定的选项启动。无论是运行 **less** 本身,还是运行另一个程序(例如 **man**)调用 **less**,情况都是如此,即都使用这些特定的选项。

如果发现使用的是 **more** 程序(假设系统上没有 **less**),那么也可以通过设置 **MORE** 环境变量,以相同的方式指定自动选项。例如,下述命令指定 **more** 总是以选项 **-cs** 启动(第一条命令针对的是 Bourne Shell 家族,第二条命令针对的是 C-Shell 家族)。

```
export MORE='-cs'
setenv MORE '-cs'
```

这两条命令也可以选择一条合适的放在登录文件中,从而使参数选项永久化。

提示

less 程序实际上查看 30 种不同的环境变量,这提供了极大的灵活性。

其中最重要的变量就是 **LESS**,这个变量我们已经讨论过。如果您对其他变量感到好奇,可以查看 **less** 的说明书页。

21.10 使用 less 显示多个文件

less 对单个文件做的任何事情都可以应用到多个文件上。特别地,您可以在文件之间来回切换,而且还可以同时在多个文件中搜索模式。出于参考目的,图 21-3 中列举了相关命令。

<code>:n</code>	切换到列表中的下一个文件
<code>:p</code>	切换到列表中的前一个文件
<code>:x</code>	切换到列表中的第一个文件
<code>:e</code>	在列表中插入一个新文件
<code>:d</code>	从列表中删除当前文件
<code>:f</code>	显示当前文件的名称(同=)
<code>=</code>	显示当前文件的名称
<code>/*pattern</code>	向前搜索指定的模式
<code>?*pattern</code>	向后搜索指定的模式

图 21-3 less: 处理多个文件的命令

为了处理多个文件，需要在命令行上指定多个文件。例如，下述命令告诉 **less** 您希望处理 3 个不同的文件：

```
less data example memo
```

在任何时候，您只能查看一个文件，这个文件就是所谓的当前文件。但是，**less** 维护一个所有文件的列表，无论何时，当需要时，都可以从一个文件切换到另一个文件。另外还可以向列表添加文件，或者从列表中删除文件。

当 **less** 启动时，当前文件就是列表中的第一个文件。在上述例子中，当前文件就是 **data**。为了在列表中向下移动，可以使用 **:n**(next, 下一个)命令。例如，如果现在读取的文件是 **data**，当键入 **:n** 时，文件就会变成 **example**，而 **example** 将成为新的当前文件。如果再次键入 **:n**，那么当前文件将变成 **memo**。

同样，在列表中还可以使用 **:p**(previous, 前一个)命令向上移动，或者使用 **:x** 命令跳到列表的开头。例如，如果现在读取的文件是 **memo**，当键入 **:p** 时，当前文件将变成 **example**。如果键入的是 **:x**，那么当前文件就变成 **data**。

为了显示当前文件的名称，可以键入 **:f**。该命令是 **=** 命令(参见图 21-2)的同义词。此时，在继续之前，您最好花点时间练习一下这 3 条命令。

less 最强大的特性之一就是允许在多个文件中搜索模式。下面介绍具体过程。

正如本章前面所讨论的，使用 **/** 命令可以在文件中向前搜索，使用 **?** 命令可以在文件中向后搜索。在使用了某个命令之后，可以键入 **n** 以相同的方向再次进行搜索，或者键入 **N** 以相反的方向再次进行搜索。

例如，假设您输入了命令：

```
/buffer
```

这将在当前文件中执行向前搜索，查找字符串“buffer”。一旦 **less** 找到了这个字符串，键入 **n** 将直接向前跳到字符串“buffer”的下一次出现。而键入 **N** 将直接向后跳到字符串“buffer”的上一次出现。

当处理不止一个文件时，您可以使用 **/*** 或 **?*** 来取代 **/** 或 **?**。当以这种方式搜索时，**less** 将把整个列表视作一个大文件。例如，假设您以前述命令启动 **less**：

```
less data example memo
```

当前文件是 **data**，文件列表是：

```
data example memo
```

键入命令：**n**，切换到列表中的第二个文件 **example**。然后键入 **50p**(50%)，移动到文件 **example** 的中间。现在输入下述命令，向前搜索字符串 “buffer”。

```
/*buffer
```

这条命令从文件 **example** 的当前位置开始，向前搜索 “buffer”。一旦搜索结束，就可以按 **n** 键向前继续搜索。如果重复按 **n** 键，那么 **less** 通常在文件的末尾处停止。但是，因为这里使用的是 **/***，而不是 **/**，所以 **less** 将自动切换到列表中的下一个文件(在本例中就是 **memo**)并继续搜索。

同样，如果重复地按 **N** 键向后进行搜索，当 **less** 到达当前文件(**example**)的开头时，它将自动地切换到前一个文件(**data**)的末尾，并继续搜索。

当使用 **/*** 替代 **/*** 执行向后搜索时，该思想同样适用。*****告诉 **less** 当使用 **n** 或 **N** 时忽略文件的边界。

除了 **n**、**p**、**x**、**/*** 和 **/*** 之外，**less** 还有两条命令帮助处理多个文件。这两条命令允许向列表插入文件及从列表中删除文件。

为了插入文件，需要键入 **:e**(**examine**，检查)，后面跟一个或多个文件名。新文件将直接插入到列表中当前文件的后面。第一个插入的文件将成为新的当前文件。例如，假设现在的文件列表为：

```
data example memo
```

当前文件是 **example**。输入下述命令向列表中插入 3 个文件：

```
:e a1 a2 a3
```

现在列表变成：

```
data example a1 a2 a3 memo
```

而当前文件现在是 **a1**。

为了从列表中删除当前文件，需要使用 **:d**(**delete**，删除)命令(当然，**less** 并不删除实际文件)。例如，如果处理的文件列表是上述列表，那么键入 **:d** 时，当前文件(**a1**)将从列表中被删除：

```
data example a2 a3 memo
```

前一个文件(**example**)成为新的当前文件。

刚开始时，这些命令可能有点混乱，特别是因为没有办法显示实际列表，所以无法知道列表是什么内容。当处理多个文件时，需要在自己的头脑中保持文件的序列。然而，当希望显示多个文件或者在多个文件中搜索时，您将会发现这些命令的实用性非常惊人，因

此它们值得学习。

21.11 使用 more 显示文件

正如本章前面所讨论的，早期的分页程序 **more** 和 **pg** 已经被功能更强大的程序 **less** 所取代。尽管您基本上永远不会看到 **pg**，但时不时地遇到 **more** 却是有可能的。例如，您可能不得不使用一个没有 **less** 的系统，这时就不得不使用 **more**。或者您使用的系统设置的默认分页程序是 **more**，您偶尔需要使用一下它^{*}。基于这些情况，您最好了解一点这个程序，因此，在本节中，我们将介绍这个程序的一些基本知识。

more 程序的语法如下：

```
more [-cs] [file...]
```

其中 *file* 是文件的名称。

more 程序每次一屏地显示数据。在写满每屏数据之后，屏幕底部的左下角会显示一个提示。提示类似于：

```
--More-- (40%)
```

(名称 **more** 就是这样得来的。)

在提示的末尾是括在圆括号中的一个数字。这个数字说明已经显示了多少数据。在我们的例子中，提示指出文件已经显示了 40%。

使用 **more** 程序最简单的方式就是指定一个单独的文件名。例如：

```
more filename
```

如果数据能在一屏上完全显示，则一次将所有的数据显示，并且 **more** 程序自动退出。否则，数据将每次一屏地显示，并且在屏幕底部显示一个提示。

一旦看到这个提示，就可以输入命令。最常见的命令就是简单地按<Space>键，这将显示下一屏数据。重复地按<Space>键，可以浏览整个文件。在显示了数据的最后一屏之后，**more** 程序将自动退出。

more 程序最常见的应用就是显示管道线的输出，例如：

```
cat newnames oldnames | grep Harley | sort | more  
ls -l | more
```

当在管道线中使用 **more** 时，提示将不显示百分比：

```
--More--
```

这是因为 **more** 在数据到达时就显示，因此它不知道会有多少数据。

^{*} Solaris 系统就是这种情况。当使用 **man** 命令显示说明书页时，默认的分页程序就是 **more**。如果您经常使用这样的系统，那么您可以通过设置 **PAGER** 环境变量，使 **less** 成为默认的分页程序。参见本章前面的讨论。

当 **more** 程序暂停时，有许多命令可供使用。像 **less** 一样，**more** 也使用原始模式(本章前面解释过)，因此，当键入单字符的命令时，不必按<Return>键。正如您所期望的，**more** 程序最重要的命令是 **h**(help, 帮助)，用来显示一个综合的命令一览表。

大多数时候，可以认为 **more** 是一种简化版本的 **less**。出于参考目的，图 21-4 列举了 **more** 程序最重要的一些命令。有关 **more** 程序的完整命令列表，请参见 **more** 的说明书页 (**man more**)。

基本命令	
h	显示帮助信息
<Space>	前进一屏
q	退出程序
高级命令	
=	显示当前行号
<Return>	前进一行
d	前进(向下)半屏
f	前进一屏
b	后退一屏
<i>/pattern</i>	向前搜索指定的模式
<i>/</i>	重复上一次搜索
<i>!command</i>	执行指定的 shell 命令
v	使用当前文件启动 vi 编辑器

图 21-4 **more**: 有用的命令

正如前面所述，按<Space>键可以向前移动一屏。另外，按 **d**(down, 向下)键可以向前移动半屏，或者按<Return>键向前移动一行。为了向后移动一屏，可以按 **b** 键(注意：**b** 命令只能在读取文件的过程中使用。在管道线中，不能向后移动，因为 **more** 不保存数据)。

为了搜索模式，只需键入`/`，后面跟希望搜索的模式，再后就是<Return>。如果愿意，还可以使用正则表达式(参见第 20 章)。当 **more** 查找到模式时，**more** 会显示这个位置的前两行，从而使您可以知道这一行的上下文。为了重复上一搜索，可以输入没有模式的`/`，也就是`/<Return>`。

在启动 **more** 程序时，有两个最有用的选项，它们是 **-s** 和 **-c**。**-s**(squeeze, 挤压)选项将多个空白行替换为一个空白行。当多个空白行没有什么特别含义时，可以使用这个选项压缩输出。当然，这不会影响原始文件。

-c(clear, 清除)选项告诉 **more** 从屏幕顶向下显示每一屏数据。每行内容在替换之前都会被清除。如果没有 **-c** 选项，新行将从屏幕的底部向上滚动。有人认为，使用 **-c** 选项读取长文件要简单一些。您可以自己试一试。

21.12 显示文件的开头: head

在第 16 章中, 我们讨论了如何在管道线中使用 **head** 作为过滤器, 从数据流的开头选取行。在本节中, 我将示范如何独立地使用 **head** 显示文件的开头。当以这种方式使用 **head** 时, **head** 程序的语法为:

```
head [-n lines] [file...]
```

其中 *lines* 是希望显示的行的数量, *file* 是文件的名称。默认情况下, **head** 显示文件的前 10 行。当希望快速地查看一个文件, 大致了解这个文件的内容时, 这条命令很有用。例如, 为了显示文件 **information** 的前 10 行, 可以使用:

```
head information
```

如果希望显示其他数量的行, 则只需使用 **-n** 选项指定一个数量。例如, 为了显示同一个文件的前 20 行, 可以使用:

```
head -n 20 information
```

提示

最初, **head** 和 **tail**(接下来讨论)不要求使用 **-n** 选项。当需要修改显示行的数量时, 只需简单地键入一个连字符, 后面跟一个数字即可。例如, 下述各个命令都显示 15 行输出:

```
calculate | head -n 15
calculate | head -15

calculate | tail -n 15
calculate | tail -15
```

正式地讲, 现代版本的 **head** 和 **tail** 都要求使用 **-n** 选项, 这就是我包含它的原因。但是, 大多数版本的 Unix 和 Linux 都接受两种类型的语法, 如果观看有经验的 Unix 人士, 就会发现他们经常省略 **-n**。

21.13 显示文件的末尾: tail

为了显示文件的末尾, 可以使用 **tail** 命令。**tail** 命令的语法为:

```
tail [-n [+]lines] [file...]
```

其中 *lines* 是希望显示的行的数量, *file* 是文件的名称。

默认情况下, **tail** 将显示文件的最后 10 行。例如, 为了显示文件 **information** 的最后 10 行, 可以使用:

```
tail information
```

为了显示其他数量的行，可以使用 **-n** 选项，后面跟一个数字。例如，为了显示 **information** 文件的最后 20 行，可以使用：

```
tail -n 20 information
```

严格地讲，必须键入 **-n** 选项。但是，正如上一节中所述，通常可以省略这个选项。也就是说，可以简单地键入一个 **-**(连字符)，后面跟希望显示的行的数量。因此，下述两行是等价的：

```
tail -n 20 information
tail -20 information
```

如果在数量前面加一个 **+**(加号)字符，那么 **tail** 将从这个行号开始，显示到文件的末尾。例如，为了从文件的第 35 行开始显示至末尾，可以使用：

```
tail -n +35 information.
```

在这个例子中，不要省略 **-n**，以确保 **tail** 不将数字解释成文件名。

21.14 观察不断增长的文件的末尾：tail -f

tail 命令拥有一个特殊的选项，允许逐行观看一个文件的增长过程。当必须等待数据写入文件时，这个选项就比较方便。例如，您可能希望监测一个向文件末尾每次写一行数据的程序。或者，如果您是一名系统管理员，那么您可能希望盯着日志文件，查看那些时不时写入日志文件的重要消息。

为了以这种方式运行 **tail**，可以使用 **-f** 选项。此时 **tail** 的语法为：

```
tail -f [-n [+]lines] [file...]
```

其中 *lines* 是希望显示的行的数量，*file* 是文件的名称(*lines* 参数在上一节中描述过)。

-f 选项告诉 **tail** 当到达文件的末尾时不要停止。相反，**tail** 要一直等下去，并且随着的文件的增长，显示更多的输出(名称 **-f** 代表“follow，跟随”)。

例如，假设在接下来的几分钟内，一个特定的程序将在文件 **results** 的末尾添加输出。您希望跟踪这个程序的运行情况，所以输入：

```
tail -f results
```

一旦输入这条命令，**tail** 将显示该文件的最后 10 行。然后 **tail** 就会等待，监视文件有没有新数据。一旦文件中添加了新行，**tail** 就会自动地显示它们。

当使用 **tail -f** 时，**tail** 一直等待新的输入，这意味着 **tail** 程序本身不会停止。为了停止 **tail** 程序，必须按下 **^C**(**intr** 键，参见第 7 章)。这可能会产生一个小问题，因为直至停止 **tail**，无法再输入任何其他命令。这种情况有两种处理方式。

首先，可以在命令的末尾使用一个 **&**(和号)字符在后台运行 **tail -f**(参见第 26 章)。


```
tail -f results &
```

当在后台运行 **tail** 时，它会悄无声息地运行，不会扰乱终端上的输出。此外，因为 **tail** 运行在您正在使用的窗口或虚拟控制台中，所以您立即会看到新的输出。这种方式的缺点是 **tail** 的输出将和您所运行的其他程序的输出混杂在一起，而这容易引起混乱。

注意：当在后台运行程序时，不能再通过 **^C** 停止程序了。相反，需要使用 **kill** 命令停止程序(细节在第 26 章解释)。

另一种方法就是使 **tail -f** 运行在自己的终端窗口或者虚拟控制台(参见第 6 章)中。如果这样做，一旦 **tail** 程序启动，就可以让 **tail** 独自运行，而您在第二个窗口或虚拟控制台中做自己的工作，并且还可以在需要时查看 **tail** 的运行情况。通过这种方式——使用两个窗口或者控制台，不仅可以在 **tail** 运行的情况下运行其他命令，而且还可以保证 **tail** 程序的输出独立。但是这种方法存在一个缺点，就是必须记住注意观察运行 **tail** 的窗口或者控制台。

如果您希望练习使用 **tail -f**，可以试试下面这个例子。首先，打开两个终端窗口(参见第 6 章)。在第一个窗口中，使用 **cat** 命令创建一个小文件 **example**：

```
cat > example
```

键入 4~5 行内容，然后按 **^D** 键结束输入，停止该命令(使用 **cat** 创建小文件的方法在第 16 章中解释过；使用 **^D**，即 **eof** 键停止命令的方法在第 7 章中解释过)。

在第二个终端窗口中，输入下述 **tail** 命令：

```
tail -f example
```

tail 程序将显示 **example** 文件的最后 10 行，然后等待新的输入。现在返回到第一个窗口，在 **example** 文件中添加一些新行。最简单的方法就是使用 **>>** 追加数据(参见第 16 章)。输入命令：

```
cat >> example
```

现在键入任意数量的行，在每行的末尾按 **<Return>** 键。注意，在第一个窗口中每键入一行，这一行就在第二个窗口中作为 **tail** 的输出显示出来。

当结束体验时，在第一个窗口中按 **^D** 键告诉 **cat** 没有输入了。然后在第二个窗口中按 **^C** 键停止 **tail** 程序。

如果回头查看使用 **tail -f** 的语法，就会发现可以指定不止一个文件名。这是因为 **tail** 可以同时监视多个文件，并且在任意文件接收到新数据时发出提醒。如果您希望体验一下多个文件的情况，可以按上一个例子中的做法建立两个终端窗口。在第一个窗口中，按上述方法使用 **cat** 创建两个小文件：

```
cat > file1
cat > file2
```

在第二个窗口中，运行下述命令：

```
tail -f file1 file2
```

现在返回到第一个窗口，轮流使用下述两条命令向两个文件中添加新行：

```
cat >> file1
cat >> file2
```

注意每次使用 **cat**(在第一个窗口中)向两个文件中的一个添加新行时，**tail**(在第二个窗口中)将显示这个文件的名称，并在后面显示新行。

21.15 二进制、八进制和十六进制

结束本章之前，我们还准备讨论两条命令 **od** 和 **hexdump**，这两条命令用来显示二进制文件的数据。为了解释这些命令的输出，需要理解二进制、八进制和十六进制计数系统。因此，在讨论这两条命令之前，我们先讨论这几个非常重要的概念。

尽管这 3 个计数系统对计算机科学和计算机编程非常重要，但是详细的讨论已经超出了本书的范围。在本节中，我们只讨论一些基本的思想。如果您热衷于成为一名计算机人士，那么我的建议是花一些时间，详细地学习一下这些主题。

大多数时候，我们使用的数由 10 个数字，即 0~9 构成。这样，我们每天使用的数都由 10 的乘幂构成：1、10、100、1000 等。这样的数字称为**十进制数**。例如，十进制数 19563 实际上是：

$$(1 \times 10000) + (9 \times 1000) + (5 \times 100) + (6 \times 10) + (3 \times 1)$$

或者使用指数表示：

$$(1 \times 10^4) + (9 \times 10^3) + (5 \times 10^2) + (6 \times 10^1) + (3 \times 10^0)$$

我们称这样的系统为**基 10 系统**或者**十进制系统**。这一名称来源的思想是，所有的数都由 10 个不同的数字构成。在计算机世界中，还有其他 3 种基，而且实际上它们比基 10 更重要：

- 基 2(二进制)：使用 2 个数字，0~1
- 基 8(八进制)：使用 8 个数字，0~7
- 基 16(十六进制)：使用 16 个数字，0~9 及 A~F

这些系统的重要性来源于计算机数据的存储方式。这是因为所有的数据都组织成电子状态序列，从概念上讲，电子状态有两种，即关或者开。对任何类型的数据来说都是如此，无论数据是停留在计算机内存(例如 RAM 或 ROM)中，还是存储在磁盘、CD、DVD、闪存或者其他设备中。

作为一种简化符号，我们使用数字 0 表示“关”，数字 1 表示“开”。通过这种方式，任何数据项，不管它长还是短，都可以认为是由 0 和 1 构成的某种模式。实际上，从技术角度而言，这就是计算机科学家考虑数据的方式：一长串的 0 和 1。

下面举几个简单的例子。在 ASCII 码中，字母“m”表示为下述模式：

01101101

单词“mellow”表示为:

011011010110010101101100011011000110111101110111

(有关 ASCII 码的讨论, 请参见第 19 章和第 20 章。有关 ASCII 码的详细列表, 请参见附录 D。)

ASCII 码只用来表示单个的字符。当需要表示数值时, 我们使用了好多种不同的系统。不考虑细节问题, 下面是使用“单精度浮点型”系统表示数 3.14159 的结果:

01000000010010010000111111010000

如果这些看起来有点混乱, 不要担心。这些细节非常复杂, 但是现在并不重要。现在重要的是理解, 对于计算机科学家来说, 所有的数据——不管数据是什么类型或者数据有多少——都存储为 0 和 1 的长串。基于这一原因, 我们将学习如何使用仅由 0 和 1 构成的数, 这样的数称为二进制数。

在计算机科学中, 一个 0 或 1 存储的数据称为一个位(bit, 代表 binary digit, 即二进制数字); 连续 8 位称为一个字节。例如, 前述例子中包含有 32 位的二进制数字或者 4 字节的数据。我们称该系统为基 2 系统或者二进制系统。该名称提醒我们, 在基 2 系统中, 所有的数都只由两个不同的数字(0 和 1)构成。

如果您是一名初学者, 那么基 2 系统的优点不会那么明显, 而且二进制数看上去没有什么意义。但是, 一旦您有经验了, 您将明白二进制数直接反应了底层数据的存储方式。这就是在许多情况下, 使用二进制数要比使用基 10 数拥有更多优点的原因。但是, 二进制也有一个问题: 二进制数不好使用, 因为它们占用大量的空间, 而且人类的眼睛很难适应它们, 容易搞混。

作为折衷, 还有两种更紧凑的, 但不丢失与底层数据直接关联的特性的二进制数表示方式。它们称为“基 8”和“基 16”。在解释它们之前, 我们先快速地示范一下如何使用基 2 计数。

在基 10 中, 我们从 0 开始计数, 直至用完数字。然后向前进位, 对左边加上数字 1, 右边又从 0 开始计数。例如, 我们从 0 开始, 计数 1、2、3、4、5、6、7、8、9, 这时用完了数字。接下来的数是 10。我们继续计数 11、12、13、14、15、16、17、18、19, 然后计数就是 20, 等等。

基 2 也采取相同的方式(实际上, 所有的基都是这样)。唯一的区别就是所使用数字的数量。在基 2 中, 只有两个数字: 0 和 1。我们从 0 开始计数, 然后是 1, 这时已用完了数字。因此, 下一个数就是 10。然后是 11、100、101、110、111、1000 等。换句话说:

0(基 10)=0(基 2)

1(基 10)=1(基 2)

2(基 10)=10(基 2)

3(基 10)=11(基 2)

4(基 10)=100(基 2)

5(基 10)=101(基 2)

请看图 21-5，该图示范了从 0 到 20 的十进制数，以及相应的二进制数(现在，可以将另外两列忽略)。

十进制(基 10)	二进制(基 2)	八进制(基 8)	十六进制(基 16)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14

图 21-5 十进制、二进制、八进制和十六进制的对应关系

在日常生活中，我们使用十进制(基 10)数。对于计算机来说，数据以二进制(基 2)数的形式存储。这样的数据可以使用八进制(基 8)或者十六进制(基 16)改写成更紧凑的形式(详情请参见正文)。

本表示范了十进制数 0 到 20 使用二进制、八进制和十六进制时的对应表示方法。您能明白该模式吗？能不能理解它们？

基 8 也称为八进制。使用这个基时，我们拥有 8 个数字，即 0~7，因此我们按下述序列计数：0、1、2、3、4、5、6、7、10、11、12 等。

基 16 也称为十六进制，与其他计数方式相似，只是使用 16 个数字。当然，如果只限于使用常规数字，则只有 10 个数字：0~9。为了在基 16 中计数，还需要 6 个数字，因此我们使用符号 A、B、C、D、E 和 F。因此，在基 16 中，我们按下述序列计数：0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F、10、11、12 等。

请再看图 21-5，到现在为止，您对全部 4 行的含义都应该有所理解了。当然，这些概念可能对您来说都是全新的，因此我并不期望您现在能够理解 3 种新的计数方式。但是，这一天会到来的，那时候这些就都简单了。例如，一名有经验的程序员在看到二进制数“1101”时，会立即知道它对应的十进制数是 13。或者当他看到八进制数“20”时，会立即知道它对应的十进制数是 16。而当看到十进制数“13”时，他又会立即知道它对应的十六进制数是 D。总有一天，您也会达到这种地步，其实这并不难，只是要多加练习。^{*}

那么为什么这非常重要呢？答案在于图 21-6。假定您有一个问题，即 3 位二进制数可以表示多少个数值呢？答案是 8 个，从 000 到 111(当包含前导 0 时)。在图 21-6 中，可以看到这 8 个值都对应一个具体的八进制数。例如，000(二进制)等于 0(八进制)，101(二进制)等于 5(八进制)，等等。这意味着任何 3 位模式(二进制数字)都对应一个单独的八进制数字。反过来，任何八进制数字也对应于一个具体的 3 位模式。

八进制(基 8)	二进制(基 2)
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

图 21-6 八进制和二进制的对应关系

每 3 个基 2 数字(位)的组合都可以由一个单独的八进制数字表示。同样，每个八进制数字也对应于一个具体的 3 位模式。

这是一个相当重要的概念，因此我们再详细讨论一下，以确保您完全理解这个概念。作为示例，考虑前面查看的“mellow”的二进制表示：

011011010110010101101100011011000110111101110111

总共有 48 位。下面将它们按 3 位分组：

011 011 010 110 010 101 101 100
011 011 000 110 111 101 110 111

^{*} 当我还是 Waterloo 大学(加拿大)的研究生时，我是学校计算机中心的系统程序员。那个时代是 IBM 大型机的天下，因此掌握十六进制算术对系统程序员来说特别重要。我们中的大多数都可以进行十六进制的加法，少数人还可以进行十六进制的减法。当然，对于更复杂的计算，我们使用计算器。但是，我的导师，他是一位令人吃惊的家伙。他叫 Romney White，他可以进行十六进制的乘法计算。Romney 是我见过的能够进行此类计算的唯一一人。

顺便说一下，现在 Romney 在 IBM 公司工作，在 IBM 公司他是在大型机上使用 Linux 的专家。当您有空时，可以在 Web 上查找他(搜索“Romney White”+“Linux”)。

通过使用图 21-6 中的表，我们可以将每 3 位一组的二进制数替换为对应的八进制数。也就是说，将 011 替换为 3，将 010 替换为 2，等等：

3 3 2 6 2 5 5 4 3 3 0 6 7 5 6 7

移除空格，得到：

3326255433067567

注意八进制比二进制紧凑多了。但是，因为每个八进制数字正好对应于 3 位(二进制数字)，所以我们完全保留了所有的信息。为什么会这么漂亮呢？因为 8 正好是 2 的 3 次幂。具体而言就是 $8=2^3$ 。因此，每个基 8 中的数字对应于基 2 中的 3 个数字。

此时，您可能想知道如果使用的计数系统基于更高的 2 的方幂，那么我们能不能用更紧凑的方式表示长字符串？答案是肯定的。这个值就是 2^4 ，即 16，使用基 16(十六进制)要比基 8 做得更好。这是因为每个十六进制数可以表示 4 位，如图 21-7 所示。

十六进制(基 16)	二进制(基 2)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

图 21-7 十六进制和二进制的对应关系

每 4 个基 2 数字(位)的组合可以由一个单独的十六进制数字表示。同样，每个十六进制数字也对应于一个具体的 4 位模式。

为了明白十六进制的原理，再次考虑表示 “mellow” 的 48 个二进制位：

011011010110010101101100011011000110111101110111

首先, 将这些位按每 4 位一组分组:

```
0110 1101 0110 0101 0110 1100 0110 1100 0110 1111 0111 0111
```

参照图 21-7, 将每 4 位一组的二进制数替换为十六进制数:

```
6 D 6 5 6 C 6 C 6 F 7 7
```

移除空格, 得到:

```
6D656C6C6F77
```

因此, 下述 3 个值是等价的。第一个值采取二进制(基 2), 第二个值采取八进制(基 8), 第三个值采取十六进制(基 16):

```
011011010110010101101100011011000110111101110111
```

```
3326255433067567
```

```
6D656C6C6F77
```

那么这意味着什么呢? 因为我们可以使用八进制字符(每个字符 3 位)或者十六进制字符(每个字符 4 位)表示二进制数据, 所以通过将二进制数据表示为八进制或者十六进制数的序列, 可以显示任何二进制文件的原始内容。实际上, 当讨论 `hexdump` 和 `od` 命令时, 我们就准备这样显示。但是, 在这之前, 我们需要先讨论另一个话题。

21.16 二进制、八进制和十六进制的读取和写入

当看到数字 101 时, 您会怎样想呢? 大多数人会说它是“一百零一”。但是, 作为计算机人士, 您可能会想: 我怎样才能知道自己正在看的数是基 10 数呢? 或许“101”指的是一个基 2(二进制)数, 而在这种情况下, 它对应的十进制数是 5:

$$(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 4 + 0 + 1 = 5$$

或许它是基 8(八进制)数, 对应的十进制数是 65:

$$(1 \times 8^2) + (0 \times 8^1) + (1 \times 8^0) = 64 + 0 + 1 = 65$$

又或许它是基 16(十六进制)数, 对应的十进制数是 257:

$$(1 \times 16^2) + (0 \times 16^1) + (1 \times 16^0) = 256 + 0 + 1 = 257$$

从上面就可以看出问题。此外, 如果希望谈论数 101, 那么它如何发音呢? 如果知道它是十进制数, 则可以以常规方式谈论它。但是如果它是二进制、八进制或十六进制呢? 这时就不能再称它为“一百零一”。

在数学中, 我们使用下标表示基。例如, 101_{16} 就意味着“基 16 的 101”, 101_8 就意味着“基 8 的 101”, 101_2 就意味着“基 2 的 101”。当没有下标时, 就是基 10 的数字。

对于计算机来说, 无法使用下标。因此, 最常见的约定就是使用一个由数字 0 和表示

基的字母构成的前缀。前缀 0x 意味着“基 16”(十六进制), 前缀 0o 意味着“基 8”(八进制), 前缀 0b 意味着“基 2”(二进制)。例如, 您可能看到 0x101、0o101 或者 0b101。有时候, 我们以一种不同的方式表示八进制, 即使用一个单独的前导 0(否则就没有必要), 例如 0101。这些约定如图 21-8 所示。

含义	数学表示法	计算机表示法	发音
基 10 的 101	101	101	“one hundred and one, 一百零一”
基 16 的 101	101 ₁₆	0x101	“hex one-zero-one, 十六进制 1-0-1”
基 8 的 101	101 ₈	0101 或 0o101	“octal one-zero-one, 八进制 1-0-1”
基 2 的 101	101 ₂	0b101	“binary one-zero-one, 二进制 1-0-1”

图 21-8 表示十六进制、八进制和二进制的约定

当使用非十进制系统时, 我们需要用来表示各个数的基的各种约定。在数学中, 我们使用下标书写这样的数。在计算机中不能使用下标, 所以我们使用一个特殊的前缀。当讨论非十进制数时, 我们对每个数字单独发音。

刚开始时, 所有这些可能有点混乱。但是, 大多数时候所使用数字的类型其实可以从上下文中猜测出。实际上, 通过观察数本身通常也可以猜测到基。例如, 如果看到的是 110101011010, 则可以猜测这个数是一个二进制数。如果看到的数像 45A6FC0, 则很容易知道这个数是一个十六进制数, 因为只有十六进制数才使用数字 A~F。

对于十六进制来说, 大写字母和小写字母都可以用作数字。例如, 0x45A6FC0 和 0x45a6fc0 是相同的。但是, 大多数人喜欢使用大写字母, 因为它们易于阅读。

当谈论各种数时, 规则比较简单。当指的是十进制数时, 只需按普通方式谈论它们即可。例如, 基 10 的 101 指的是“一百零一”; 3056 指的是“三千零五十六”。

至于其他基, 我们只需简单地说出每个数字的名字, 有时候还需要提到基。例如, 如果希望讨论基 16 的 101, 则可以说“1-0-1 基 16”或者“十六进制 1-0-1”; 基 16 的 3056 就是“3-0-5-6 基 16”或者“十六进制 3-0-5-6”。

同理, 基 8 的 101 是“1-0-1 基 8”或者“八进制 1-0-1”; 基 2 的 101 是“1-0-1 基 2”或者“二进制 1-0-1”, 或者其他类似的东西。图 21-8 中描述了这些发音。

21.17 选择使用十六进制而不是八进制的原因

在第 7 章中, 我们讨论过以前创建计算机存储器所使用的技术类型。特别地, 我提到过在 20 世纪 50 年代以及 20 世纪 60 年代, 存储器都是由微小的磁芯组成的。基于这一原因, 单词 core 成为存储器的同义词。

在那个时代, 程序的调试非常困难, 特别是当程序非正常终止时。为了帮助解决这样的问题, 程序员可以指示操作系统打印程序终止时刻所使用的存储器内容。然后程序员就可以研究打印的数据, 尝试推断发生了什么问题。正如第 7 章中解释的, 这样的数据称为磁芯转储(core dump), 而且解释这样的数据需要许多技能。现在, “磁芯转储”这一表示方

法仍在使用，只是有时候被内存转储(memory dump)或者转储(dump)所替代。

20 世纪 70 年代初期，在开发 Unix 的过程中，调试异常困难，程序员经常不得不保存并检查转储。尽管技术已经进步——磁芯已经被半导体替代，但是人们仍然将存储器称为磁芯，而存储器内容的副本仍然称为磁芯转储。因此，当 Unix 将存储器的内容保存到文件，用于稍后检查时，文件仍被称为磁芯文件(core file)，该文件的默认名称就是 **core**。

因此，从一开始，Unix 程序员就需要检查磁芯文件。为了满足这一需求，Unix 开发人员创建了一个以八进制(基 8)数字显示磁芯文件内容的程序。这个程序名为 **od(octal dump, 八进制转储)**，而且多年以来，它已被证明是一个特别有用的程序。即使是今天，使用 **od** 仍然是查看二进制数据的最佳方式之一。

正如前面所讨论的，二进制数据既可以表示为八进制数(每个数字使用 3 位)，也可以表示为十六进制(每个数字使用 4 位)。八进制相对容易学习，因为它使用的数字(0 至 7)我们已经熟悉。而十六进制要求使用 16 个数字，其中 6 个数字(A、B、C、D、E、F)并不属于我们的日常文化。这样，十六进制就比八进制更难学。然而，十六进制比八进制的应用要多，原因有以下 3 方面。

首先，十六进制比八进制更加紧凑(准确地讲，十六进制要比八进制紧凑三分之一)。如果要显示各个位，相对于八进制，十六进制需要更少的字符就可以完成任务。

十六进制更流行的第二个原因在于计算机处理位的方式。计算机处理器将位组织成基本单元进行管理，这种基本单元称为字(word)，字的大小取决于处理器设计。20 世纪 60 年代中期，大多数处理器都使用 16 位或者 32 位的字长；现在，处理器通常使用 64 位的字长。在 20 世纪 50 年代和 60 年代，还有许多计算机，特别是科学计算机，使用 24 位或 36 位的字长。

对于 24 位或 36 位字长来说，既可以使用八进制系统，也可以使用十六进制系统，因为 24 和 36 都能被 3 和 4 整除。因为八进制比较简单，所以在 20 世纪 50 年代和 60 年代被广泛使用。

对于 16 位、32 位和 64 位字长来说，使用八进制系统就很困难了，因为 16、32 和 64 不能被 3 整除。但是，可以使用十六进制系统，因为 16、32 和 64 都可以被 4 整除。基于这一原因，自 20 世纪 70 年代起，十六进制的应用越来越多，而八进制的应用越来越少。

十六进制应用广泛的第三个原因就是尽管十六进制比较难学，但是一旦学会了，十六进制更容易使用。基于这一原因，即便是最早版本的 **od** 程序也提供了一个选项来以十六进制显示数据。

正如前面所述，受人尊敬的 **od** 程序已经面世许多年了，实际上，从 Unix 开始就有了。但是，在 1992 年，另一个这样的程序 **hexdump** 面世，它是为 BSD(伯克利 Unix，参见第 2 章)编写的。现在，**hexdump** 应用广泛，不仅安装在 BSD 系统上(例如 FreeBSD)，还作为众多 Linux 发行版的一部分随 Linux 一起发行。

大多数有经验的 Unix 人士倾向于选择一个自己喜欢的程序使用，要么是 **od**，要么是 **hexdump**。基于这一原因，我准备示范两个程序的使用方法，以使您可以看看自己喜欢哪一个程序。

21.18 显示二进制文件：hexdump、od

hexdump 和 **od**(octal dump, 八进制转储)最初的应用都是查看包含在磁芯文件中的内存转储。通过查看转储,程序员可以跟踪那些用其他方式难以捕捉的 bug。现在,已经出现了更好的调试工具,程序员也极少手工查看磁芯文件了。但是, **hexdump** 和 **od** 仍然有用,因为它们可以以可读的格式显示任何类型的二进制数据。实际上,这两个程序是两个基于文本的查看二进制文件内部的主要工具。

因为这两个程序都可以完成任务,所以我将同时说明这两个程序的使用方法。然后您可以动手体验一下,看看喜欢哪一个。两个程序之间最大的区别就是 **hexdump** 默认情况下以十六进制显示数据,而 **od**(比较老一些)默认情况下以八进制显示数据。因此,如果使用 **od**,您必须记住那些可以生成十六进制输出的选项。

另一方面, **od** 在全部 Unix 系统上可用,而 **hexdump** 不是这样。例如,如果您使用的是 Solaris,那么您可能没有 **hexdump**。基于这一原因,处理二进制文件时,即使喜欢使用 **hexdump**,也应该知道一些 **od** 的知识,以防有一天不得不使用它。

在介绍语法之前,先看一些典型的输出。在图 21-9 中,可以看到存放 **ls** 程序的文件中的一部分二进制数据(**ls** 程序用来列举文件名,我们将在第 24 章中讨论)。

OFFSET	HEXADECIMAL																ASCII
000120	00	00	00	00	00	00	00	00	00	00	00	06	00	00	00	00
000130	04	00	00	00	2f	6c	69	62	2f	6c	64	2d	6c	69	6e	75	.../lib/ld-linu
000140	78	2e	73	6f	2e	32	00	00	04	00	00	00	10	00	00	00	x.so.2.....
000150	01	00	00	00	47	4e	55	00	00	00	00	00	02	00	00	00	...GNU.....
000160	06	00	00	00	09	00	00	00	61	00	00	00	76	00	00	00a...v...
000170	00	00	00	00	4c	00	00	00	4b	00	00	00	3f	00	00	00	...L...K...?..

图 21-9 以十六进制和 ASCII 码显示的样本二进制数据

可以使用 **hexdump** 或 **od** 命令显示二进制数据。这里以规范格式显示了这类数据的一个样本。也就是说,左边是十六进制的偏移,中间是十六进制的数据,右边是数据的 ASCII 字符。这个特殊的例子取自包含 GNU/Linux 版 **ls** 程序的文件。

当检查文件中的数据时,有时候需要知道所查看内容的准确位置。当使用 **less** 查看文本文件时,可以方便地推断出在什么位置。在任何时候,可以使用=(等号)命令显示当前行号。另外,还可以使用-M 选项在提示中显示当前行号,或者使用-N 选项在每行的左边显示一个行号。

对于二进制文件而言,文件中没有行,因此行号没有意义。作为替代,我们使用偏移(offset)标记每个位置。偏移就是一个数字,告诉离文件开头有多少字节。其中,第一个字节的偏移是 0;第二个字节的偏移是 1;依此类推。

请看图 21-9 中的样本数据。偏移位于最左边一栏,它不属于数据。在我们的例子中,所有数字都是十六进制,因此数据第一个字节的偏移是 0x120(也就是十六进制 120),或者是十进制的 288。因此,例子中的第一个字节是文件的第 289 个字节(记住,偏移从 0 开始)。

输出的第一行包含 16 个字节。因此,其偏移从 0x120 到 0x12F。第二行从偏移 0x130 开始。每个开始偏移的右边是 16 字节的数据,以两种不同格式显示。其中中间栏是表示数

据的十六进制数字，按字节分组(记住，1 字节=8 位=2 个十六进制数字)。右边，相同的数以 ASCII 字符显示。

在大多数二进制文件中，有一些字节包含的是实际 ASCII 字符。通过查看最右边的一栏可以方便地识别这些字节。在上面的例子中，可以看到字符串 `/lib/ld-linux.so.2` 和 `GNU`。根据约定，没有对应于可显示 ASCII 字符的字节一律用一个.(句点)字符表示。在上面的例子中有许多这样的字节。

二进制文件中的大多数字节都不是字符，而是机器指令、数值数据等。通过查看最右边一栏就可以分辨出是不是字符，在最右边的栏中，大多数都是.标记，只有少数几个字符点缀在其间。在我们的例子中，第一行和最后两行包含的都是非 ASCII 数据。少数几个包含值的字节碰巧对应于字符，但是这只是巧合，并没有实际意义。

图 21-9 中数据显示的方式称为规范格式(canonical format)。该格式用于二进制数据的显示或打印，每行包含 16 字节。每行的左边是以十六进制表示的偏移。中间是实际数据，也采用十六进制表示。右边是对应的 ASCII 值。

`hexdump` 和 `od` 都能以多种不同格式显示二进制数据。实际上，两个命令都支持大量的选项，从而使您可以获得对数据显示方式的更多控制。但是，大多数时候，最好是使用规范格式。基于这一原因，在讨论这些命令时，我将示范使用哪些选项可以生成这种类型的输出。如果需要其他变体的信息，可以查看相应的说明书页。

我们首先从 `hexdump` 开始，因为它使用起来比较简单。在使用 `hexdump` 以规范格式显示一个二进制文件时，其语法比较简单：

```
hexdump -C [file...]
```

其中 *file* 是文件的名称。

`hexdump` 程序有许多选项，允许控制输出。此外，`hexdump` 还有一个重要的快捷方式：如果使用 `-C`(canonical, 规范)选项，那么 `hexdump` 将自动使用合适的选项组合，生成规范输出。

下面举例说明。假设希望查看包含 `ls` 程序的二进制文件的内部。首先，使用 `whereis` 程序查找这个文件的路径名，也就是这个文件的准确位置(我们将在第 24 章中讨论路径名和 `whereis`，因此现在不用关心细节问题)。所使用的命令为：

```
whereis ls
```

典型的输出为：

```
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

该输出显示了这个程序及其说明书页的准确位置。我们只对该程序感兴趣，因此使用第一个路径名：

```
hexdump -C /bin/ls | less
```

该命令以规范格式显示整个文件的内容。这一切就是这么简单。

如果希望限制显示数据的数量，则有两个选项可以使用。`-s(skip over, 略过)`选项允许

通过设置初始偏移指定在文件开头略过多少字节。例如，为了从偏移 0x120(十六进制 120)处开始显示数据，可以使用：

```
hexdump -C -s 0x120 /bin/ls | less
```

为了限制输出量，还可以使用 **-n**(number of bytes, 字节数量)选项。下述命令从偏移 0x120 处开始显示，并且只显示 96 字节的数据(也就是说，显示 6 行输出)。在这个例子中，输出量非常小，因此不用管道传送给 **less** 程序：

```
hexdump -C -s 0x120 -n 96 /bin/ls
```

顺便说一下，这就是生成图 21-9 中所显示输出的命令。

将这两个选项集成到语法中，我们可以为 **hexdump** 定义一个更全面的语法：

```
hexdump -C [-s offset] [-n length] [file...]
```

其中 *file* 是文件的名称，*offset* 是在文件开头略过的字节数，*length* 是要显示的字节数。注意：**offset** 可以使用任何基，但是 **length** 必须是十进制数(没人知道为什么，自己虚构原因吧)。

提示

对于 FreeBSD 来说，可以将命令 **hd** 作为 **hexdump -C** 的别名使用。因此，在 FreeBSD 系统上，下述两条命令是等价的：

```
hd /bin/ls
hexdump -C /bin/ls
```

如果希望在不同的系统上使用这条方便的命令，那么您需要使用下述命令之一自己创建一个别名。第一条命令针对的是 Bourne Shell 家族，第二条命令针对的是 C-Shell 家族。

```
alias hd='hexdump -C'
alias hd 'hexdump -C'
```

为了使别名永久化，可以在环境文件中放入一条合适的命令(别名在第 13 章讨论过，环境文件在第 14 章讨论过)。

使用 **od** 以规范格式显示二进制文件时，**od** 程序的语法为：

```
od -Ax -tx1z [file...]
```

其中 *file* 是文件的名称。

-A(address, 地址)选项允许指定使用哪一种计数系统表示偏移值。对于规范输出来说，需要将其指定为 **x**，这将以十六进制显示偏移。

-t(type of format, 格式类型)选项控制数据显示的方式。对于规范输出来说，需要将其指定为 **x1**(以十六进制显示数据，每次一个字节)和 **z**(在每行末尾显示相应的 ASCII 值)。有关格式代码的完整列表，请参见说明书页(**man od**)或者信息文件(**info od**)。

(注意：该语法针对 GNU 版本的 **od** 程序，例如在 Linux 系统中使用的 **od** 程序。如果

您使用的系统没有 GNU 实用工具，那么该命令将更原始。比如说，您不能使用 **z** 格式代码。详情请查看说明书页。)

作为示例，下述 **od** 命令与前面所示 **hexdump** 命令等价。它以规范格式显示 **ls** 文件的内容：

```
od -Ax -tx1z /bin/ls | less
```

如果希望限制显示的数据量，有两个选项可以使用。**-j**(jump over, 跳过)选项指定从文件开头跳过多少字节。例如，为了从偏移 0x120(十六进制 120)处开始显示数据，可以使用：

```
od -Ax -tx1z -j 0x120 /bin/ls | less
```

为了限制输出量，还可以使用**-N**(number of bytes, 字节数量)选项。下述命令从偏移 0x120 开始，并显示 96 字节数据(6 行输出)。在这个例子中，输出量非常小，因此没有必要将输出管道传送给 **less** 程序：

```
od -Ax -tx1z -j 0x120 -N 96 /bin/ls
```

该命令生成和图 21-9 中所示相似的输出。

将这两个选项集成到语法中，可以为 **od** 定义一个更全面的语法：

```
od -Ax -tx1z [-j offset] [-N length] [file...]
```

其中，*file* 是文件的名称；*offset* 是在文件开头略过的字节数；*length* 是要显示的字节数，可以使用十进制、十六进制或八进制指定。

提示

od 程序的语法比较复杂、笨拙。但是，可以通过创建别名并指定以规范格式生成输出的选项而进行简化。下述两条命令都可以实现该目的。其中第一条命令针对的是 Bourne Shell 家族，第二条命令针对的是 C-Shell 家族。

```
alias od='od -Ax -tx1z'
alias od 'od -Ax -tx1z'
```

一旦定义了这样的别名，当键入 **od** 时，就会自动地获得希望的输出。为了使别名永久化，可以在环境文件中放入一条合适的命令(别名在第 13 章讨论过，环境文件在第 14 章讨论过)。

名称含义

canonical(规范)

在本章前面，当讨论基于文本的交互式程序如何处理输入时，我们介绍过规范模式和非规范模式。在本节中，我们又提到某特定格式的二进制输出称为规范输出。计算机科学家所使用的“canonical”与其常规的英语含义不同，因此要理解这之间的区别。

在通用英语中，单词“canonical”指的是一种教规思想，即一组管理基督教堂成员的正式规则。canonical 描述遵循教规的事项，因此，可以引申为天主教堂教规程序。

在数学中，该术语拥有更明确更现代化的含义。它指的是用最简单、最重要的方式表达数学思想。例如，高中生所学习的解二次方程的规范公式。

计算机科学家从数学借来了这一术语，并且极大地扩展了它的含义。在计算机科学中，canonical 指完成事情的最常见的习惯方式。例如，在介绍 `hexdump` 和 `od` 命令时，我们讨论了显示二进制数据的规范格式。这种格式没有神奇之处。但是，它非常适用，这种格式已经使用了 40 多年，而且它就是人们所期望的，所以它成为了规范。

21.19 众多计算机术语来自数学的原因

随着对计算机科学的学习越来越深入，您将会发现有许多计算机术语派生自数学。作为示例，单词“canonical”——我们在本章以两种不同的方式使用了这个术语，就是来自一个相似的数学术语。您可能会奇怪为什么有如此众多的计算机术语来自数学。原因有以下几方面。

早期的计算机科学于 20 世纪 50 年代至 60 年代由创建于 20 世纪 30 年代和 40 年代的数学理论发展而来。实际上，计算机科学的数学基础主要来源于 Alan Turing(1912-1954)、John von Neumann(1903-1957)、Alonzo Church(1903-1995)以及(较少)Kurt Gödel(1906-1978)等人的工作。

在 20 世纪 50 年代和 60 年代期间，几乎所有的计算机科学家都是数学家。实际上，计算机科学被认为是数学的一个分支^{*}。因此，计算机先锋们从原先的领域中获取描述新思想的术语非常自然。

多年以来，随着计算机科学的发展，它要求大量的分析和形式化。和其他科学一样，计算机科学所需的技术和见解都取自数学，这是因为数学已经研究和形式化了 2000 多年，相关的工具非常丰富(这就是伟大的德国数学家和科学家 Carl Friedrich Gauss 称数学为“科学之王”的原因)。即使是现在，计算机科学家和程序员也需要主要来自数学的抽象和逻辑推理。这样做时，他们很自然地会修改数学术语以适应自己的需要。

提示

数学和计算机的关系就如希腊语和拉丁文与英语的关系一样。

21.20 练习

1. 复习题

1. 哪些程序可以用来每次一屏地显示文本文件？一次显示整个文本文件？显示文本

^{*} 在我做本科生课题的学校，即加拿大 Waterloo 大学，计算机科学系当时(现在仍然)属于数学学院。实际上，我的大学学位就是主修计算机科学的数学学士。

文件的最初部分？显示文本文件的最后部分？显示一个二进制文件？

2. 在使用 **less** 显示文件时，使用哪些命令执行下述动作：前进一屏；后退一屏；跳到第一行；跳到最后一行；向前搜索；向后搜索；显示帮助；退出程序？

3. 可以使用 **less** 程序显示不止一个文件，例如：

```
less file1 file2 file3 file4 file5
```

在阅读文件的过程中，使用哪些命令执行下述动作：切换到下一个文件；切换到上一个文件；切换到第一个文件；从列表中删除当前文件？哪些命令用来在所有文件中向前搜索和向后搜索？

4. 使用哪条命令可以查看不断增长的文件的末尾？

5. 当显示二进制文件时，什么是规范格式？如何使用 **hexdump** 以规范格式显示文件？使用 **od** 呢？

2. 应用题

1. 检查 **PAGER** 环境变量的值。如果它没有被设置为 **less**，将其设置成 **less**。显示 **less** 程序本身的说明书页(参见第 9 章)。执行下述操作：

- 显示帮助信息。每次一屏地向下移动帮助信息，并阅读每屏信息。退出帮助。
- 向前搜索“help”。再次搜索。再次搜索。换为向后搜索。
- 跳到说明书页的末尾。
- 后退一屏。
- 跳到第 100 行。
- 显示当前行号。
- 跳到说明书页的 20%处。
- 显示当前行号。
- 跳到说明书页的开头。
- 退出。

2. 希望使用 **less** 显示文件 **list** 的内容，该文件中包含许多文本行。在显示文本时，还希望同时显示行号。但是，文本中没有行号，而且不希望以任何方式改变原始文件。如何使用 **nl** 和 **less** 实现该目标？只使用 **less** 如何实现该目标？使用 **nl** 有何优点？

3. 将下述二进制(基 2)数转换成八进制(基 8)、十六进制(基 16)和十进制(基 10)数。必须示范转换过程。

```
1111101000100101
```

4. 使用 **strings** 命令(参见第 19 章)查找二进制文件 **/bin/ls** 中的字符串。选取一个字符串，使用 **hexdump** 或 **od** 查找这个字符串在文件中的准确位置。

3. 思考题

1. 希望使用 `less` 搜索 5 个不同文件，查找一个特定的单词序列。最明显的解决方法就是使用 `less`：

```
less file1 file2 file3 file4 file5
```

但是，这要求追踪并操作 5 个不同的文件。作为替代，可以先将 5 个文件组合成一个大文件：

```
cat file1 file2 file3 file4 file5 | less
```

这样会使这个工作简单些还是困难些？为什么呢？

2. 在正文中，我们讨论了 4 种计数系统：十进制(基 10)、二进制(基 2)、八进制(基 8)和十六进制(基 16)。原则上，任何正整数都可以用作基，例如基 12，也可以称为十二进制(duodecimal)。在基 12 中，使用的数字从 0 至 9，另外两个额外数字使用 A 和 B 表示。因此，在基 12 中，我们按如下顺序计数：1、2、3、4、5、6、7、8、9、A、B、10、11、12，依此类推。基 10 中的数 22 等价于基 12 中的 1A。

相对于基 10，基 12 计数有何重要的优点？相对于 10，12 有几个整除因子？基 12 如何简化计算？如果使用基 12 取代基 10，我们的文化会不会好些？

不管基 12 有什么优点，我们都不会在计算机中使用基 12。大多数时候我们使用基 16，而基 16 更复杂。为什么会这样呢？

顺便说一下，如果您曾经阅读过 J.R.R. Tolkein 所著的 *Lord of the Rings* 一书，那么您就会知道 Elvish 语言使用的就是十二进制计数系统。



vi 文本编辑器

文本编辑器(通常称为**编辑器**)就是一个用来创建和修改文本文件的程序。当您使用这样的程序修改文件时,我们称之为**编辑文件**。

学习如何熟练地使用 Unix 编辑器非常重要,因为处理纯文本文件需要使用编辑器。如果您是一名程序员,那么您还需要使用编辑器编写程序。对于非编程任务来说,需要使用编辑器创建和修改配置文件、shell 脚本、初始化文件、Web 页面、简单文档等。实际上,当需要处理任何包含文本的文件时,都需要使用编辑器。

与字处理程序不同,编辑器只处理纯文本,也就是说由可显示字符构成的数据,包括字母、数字、标点符号、空格和制表符(参见第 19 章)。通常,编辑器使用一种简单的等宽字体。因此,当希望创建一个拥有多种字体的文档,或者需要使用各种字体大小、颜色、斜体、粗体或者其他属性时,不能使用编辑器。对于这样的工作,应该使用字处理程序。

在本章中,我们将讨论 **vi**,它是最重要的 Unix 文本编辑器。尽管我不能解释所有的内容——这样的话至少需要好几章,但是我会介绍大多数时候最需要知道的内容(名称 **vi** 的发音是两个单独的字母:“vee-eye”)。

本章自始至终将讨论“文件”的编辑,但是严格地讲,我们还没有真正解释文件到底是什么。在第 23 章中,当讨论 Unix 文件系统时,将对文件进行解释。现在,我们只需使用直觉印象,即认为文件具有一个名称,并包含信息。例如,可以使用 **vi** 编辑文件 **essay**,而 **essay** 包含有您所写论文的文本。

当第一次学习 **vi** 时,**vi** 看上去难以使用。这没有办法逃避。如果您感觉受到挫折,那么请记住,每个人第一次学习 **vi** 时都会有相同的感受。尽管如此,一旦您成为有经验的用户,所有事情都会变得很简单,那时候 **vi** 看上去将是那么的自然而又容易使用。这就意味着下面的提示是正确的,在第 1 章中,我们已经将其应用在 Unix 上了。

提示

vi 用起来容易,但学习起来难。

22.1 vi 重要的原因

Unix/Linux 文本编辑器有许多种,包括我们在第 14 章中讨论的简单文本编辑器(**kedit**、

gedit、Pico 和 Nano)。但是，两个主要的 Unix 文本编辑器是 **vi** 和 Emacs，两种文本编辑器都已经面世很长一段时间了。它们都是强大、成熟、功能完整的程序，到目前为止，它们都是使用最广泛的编辑器。这两个编辑器，不管哪一个都可以满足需求。

vi 和 Emacs 两种编辑器之间存在极大的区别，可以想象，会有很多人坚持认为其中一个比另一个更优秀。实际上，Unix 社区许多年来一直在争论这个问题。事实是 **vi** 和 Emacs 代表文本编辑的完全不同的方式。这样，就产生了 **vi** 的用户群和 Emacs 的用户群，当说起编辑时，他们以不同的角度来看待。最终，当您拥有足够的经验时，您也不得不进行选择：是做 **vi** 人士呢还是做 Emacs 人士呢？

现在，您只需理解 **vi** 和 Emacs 都非常复杂，它们都需要花很长的时间学习。在本章中，我将讲授如何使用 **vi**，因为它是两个编辑器中更重要的一个。以后，如果需要，您可以自学 Emacs。

无论最终选择使用哪个程序作为自己的主要编辑器，您都必须学习 **vi**。原因在于，与其他任何编辑器不同，**vi** 无处不在。有许多人在使用 **vi**，并且使用了相当长的时间，而且在世界上几乎每个 Unix 和 Linux 系统上都能找到它。更正式地讲，**vi** 属于两个主要的 Unix 规范：POSIX(参见第 11 章)和单一 Unix 规范(Single Unix Specification, 如果要称之为“Unix”必须满足的标准)。因此，根据定义，不管有多么难以安排，**vi** 都必须位于每个 Unix 系统上。这意味着，一旦知道了如何使用 **vi**，就可以在任何可能遇到的 Unix 或 Linux 系统上编辑文本，而后者又意味着能够编辑配置文件、创建初始化文件或者编写一个简单的 shell 脚本。

当在一个提供有限工具的环境中工作时，这显得特别重要。例如，如果系统出现了问题，从救援磁盘启动后，您或许会发现 **vi** 是唯一可用的文本编辑器。同样，**vi** 通常是嵌入式系统(如移动电话、DVD 播放器、仪器上的计算机化设备)上唯一可用的编辑器。

就个人而言，我从 1976 年开始使用 Unix，还从来没有见过哪一个系统没有提供 **vi***。实际上，**vi** 编辑器的应用如此之广，而且如此之重要，如果您申请一个 Unix 或 Linux 方面的工作，招聘方将假定您知道使用 **vi**。

22.2 vi 历史简介

vi 编辑器由 Bill Joy 创建，当时他是加利福尼亚大学伯克利分校 20 世纪 70 年代末的一名研究生(参见图 22-1)。Joy 是一名令人惊讶的高水平程序员，是早期 Unix(参见第 2 章)最多产、最重要的贡献者之一。除 **vi** 之外，Joy 还负责原始 BSD(伯克利 Unix)、C-Shell 以及 TCP/IP 支持 Internet 的协议的第一个健壮实现的开发。1982 年，Joy 与他人合作创建了 Sun 公司，在 Sun 公司中，Joy 对 NFS(Network File System, 网络文件系统)和 SPARC 微处理器体系结构的开发做出了极大的贡献。为了理解在 20 世纪 70 年代末，Joy 作为一名研究生编写 **vi** 的动机，我们有必要返回到 Unix 的早期年代。

* 一些 Linux 发行版没有自动地安装 **vi** 或者 Emacs。最常见的原因是发行版的创建者不愿意在永无终止的 **vi**/Emacs 争论上袒护哪一方。作为折衷，它们默认安装一个较简单的文本编辑器，例如 Nano(参见第 14 章)。放心，当遇到这样的系统时，安装 **vi** 非常简单。



图 22-1 Bill Joy 和 Dennis Ritchie

在 20 世纪 70 年代末, Bill Joy(右)是加利福尼亚大学伯克利分校的一名研究生。在那段时间, Joy 创建了 **vi** 编辑器和 C-Shell, 并且组建了第一版的 BSD(伯克利 Unix)。

这张照片拍摄于 1984 年 6 月, 摄于在犹他州 Snowbird 举办的 Usenix 会议上。和 Joy 合影的是 Dennis Ritchie(左), 他是原始 Unix 操作系统(参见第 3 章)的开发人员之一。在拍摄这张照片时, BSD、**vi** 和 C-Shell 已经在世界范围广泛使用。Joy 左袖子上的圆中(在您的右边)写着 “The Joy of Unix”。

第一个重要的 Unix 编辑器是 **ed**(发音是两个单独的字母 “ee-dee”)。**ed** 编辑器由 Ken Thompson 于 1971 年在贝尔实验室编写, Ken Thompson 是 Unix 的两名最初创建者之一(另一位是 Dennis Ritchie, 参见第 1 章和第 2 章)。**ed** 编辑器是一种面向行的编辑器(line-oriented editor), 或者叫行编辑器(line editor), 这意味着它处理编号的文本行。例如, 可以输入命令打印第 100 行至第 150 行, 或者删除第 17 行。这种方法非常有必要, 因为早期的终端非常慢, 而且处理能力有限(参见第 3 章)。

在 1975 年秋天, Bill Joy 离开 Michigan 大学, 成为加利福尼亚大学伯克利分校的一名计算机科学研究研究生。他准备学习计算理论, 这个专业相当偏向数学。但是, 他的工作致使他转向编程, 并且还是面世不久的 Unix。原因是 Ken Thompson 这段时间碰巧也在伯克利。

Thompson 正好在贝尔实验室休假一年, 他决定这一年在自己的母校伯克利分校度过, 于是作为一名访问教授回到母校。他到达的时候, 正好伯克利计算机科学系得到了一台全新的 PDP 11/70 微型计算机。和两个学生一起, Thompson 在这台新计算机上安装了最新版本的 Unix(版本 6)。然后, 他在 Unix 中安装了一个 Pascal 系统(Pascal 是一种编程语言, 创建于 1970 年, 由瑞士计算机科学家 Niklaus Wirth 创建, 用来讲授结构化编程)。

Joy 和另一名学生 Charles Haley 对 Thompson 的 Pascal 系统产生了浓厚的兴趣, 他们决定使用它开发一个与上下文无关的通用语法分析算法(用来分析源程序结构的方法)。但是, Joy 和 Haley 很快发现 Pascal 系统拥有极大的局限性。他们开始修复问题, 这使得他们开始责备原始的 **ed** 编辑器。

Joy 和 Haley 发现 **ed** 非常不方便, 所以他们决定创建一个更好的编辑器。那时, 另一名访问者, 来自 Queen Mary 学院(伦敦)的 George Coulouris 将他自己的软件带到了伯克利: 一个叫 **em**(发音为 “ee-em”)的编辑器。Coulouris 创建了 **em**, 作为 **ed** 的一个向后兼容的替代品。他选择 **em** 作为名称, 含义是 “editor for mortals, 人类编辑器”(也就是说 **ed** 并不适合正常的人类)。Joy 和 Haley 将 **em** 的部分功能集成到 **ed** 中, 创建了一个混合体, 他们称之为 **en**(发音为 “ee-en”)。

en 编辑器还远没有达到完美。Joy 和 Haley 花费了大量的时间改进 **en**, 创建了一个又

一个新的版本。最终，他们实现了一个相当出色的版本，称为 **ex**(发音为“ee-ex”)。与 **ed** 相比，**ex** 是一个有了极大改进的编辑器。然而，**ex** 仍然是一个面向行的编辑器，这种类型的程序适用于非常古老的终端和缓慢的调制解调器。

但是到了 1976 年，Joy 和 Haley 接触到一种新型的终端，即 Lear Siegler ADM-3A 终端。ADM-3A 要比贝尔实验室中用来开发 **ed** 的古老的 Teletype ASR33(参见第 3 章)高级得多。ASR33 将输出每次一行地打印在纸上，而 ADM-3A 有一个显示器，能够在屏幕的任何地方显示文本。图 22-2 为 Lear Siegler ADM-3A 终端的一张照片。将这张照片与第 3 章中 Teletype ASR33 的照片对比一下，就会知道它们之间的区别。



图 22-2 Lear Siegler ADM-3A 终端

计算机领域有一个著名的原则，即软件的增强由新硬件驱动。1976 年，当 Bill Joy 开发 **vi** 文本编辑器时，情况就是这样。较古老的面向行的编辑器，例如 **ed** 和 **ex**，都是为原始的面向行的终端设计的。但是，当 Joy 开始接触一种新型的面向屏幕的终端——Lear Siegler ADM-3A 终端时，该终端的先进功能激发 Joy 创建了 **vi**，即一种面向屏幕的编辑器。详情请参见正文。

为了利用 ADM-3A 所提供的能力，Joy 通过创建一个独立的面向屏幕的界面增强了 **ex**，他称这个界面为 **vi**(发音为“vee-eye”)。新的 **vi** 编辑器支持所有的 **ex** 命令，以及大量的新命令，这些新命令允许使用完整的屏幕。例如，与相对古老的面向行的编辑器不同，**vi** 允许在编辑文件的过程中，直观地从一个位置跳到另一个位置。另外，还可以插入、修改或删除任何位置的文本，而不必担心行号。通过这种方式，**vi** 成为一种面向屏幕的编辑器(或者屏幕编辑器)。

时至今日，**vi** 编辑器仍然是面向屏幕的命令和面向行的命令的混合体。因此，在学习 **vi** 的过程中，必须自学两种不同类型的命令。可以想见，这意味着要想学会熟练地使用 **vi**，需要花费一段时间。然而，另一方面，能够同时以两种不同的方式编辑数据，使 **vi** 成为一个特别强大的工具。

更有趣的是，现在的 **vi** 和 **ex** 实际上是同一个程序。如果使用 **vi** 命令以普通方式启动程序，则看到的是面向屏幕的界面。如果使用 **ex** 命令启动程序，则得到的是较古老的面向行的界面。

名称含义

ed、**ex**、**vi**

在 Unix 的初期，许多命令的名称都是很短的，只包含两个字母。对于这些名称，传统是以两个单独的字母发音。例如，**ed** 的发音是“ee-dee”，**ex** 的发音是“ee-ex”，而 **vi** 的发音是“vee-eye”。将 **vi** 发音为一个单音节的“vie”是不正确的。

ed 的含义比较简单，它代表“editor，编辑器”。

ex 的含义没有那么直接。许多人认为选择这个名称，是因为它意味着“extended editor，扩展编辑器”。从某种意义上讲，这没有错，因为 **ex** 就是 **ed** 功能的极大扩展。实际上，尽管细节可能有点模糊，但是这一名称确实是模式 **ed... en... ex** 的延续。**ex** 的合作开发者 Charles Haley 曾经向我这样解释：“我记得有过一个 **en**。但我记不起有 **eo** 了。我想我们应该是从 **en** 到‘e-什么’或者 **ex**。”Bill Joy 在一次访谈中也有类似的说明：“我不知道是否有过一个 **eo** 或一个 **ep**，但是最后出现了一个 **ex**。我记着有 **en**，但是我不知道它是如何变到 **ex** 的。”

如果您对仍在使用的两个字母的命令名称感兴趣，可以查看第 20 章末尾，那里示范了如何使用 **grep** 程序在系统中查找这些名称。如果有时间，还可以在联机手册中查找这些命令：您将会发现一些已被忘记的珍贵命令。

不管怎么样，当 Joy 通过添加面向屏幕的命令扩展 **ex** 时，他选择了一个新的两个字符的名称：**vi**。**vi** 的含义是“visual editor，可视编辑器”。

如此众多的命令被命名为两个字符的名称，其实际原因有两方面。首先，聪明的人倾向于选择短的、易于使用的缩写。其次，老式的终端运行速度慢得令人苦恼，使用短命令易于正确地键入命令。

22.3 Vim: vi 的备用编辑器

vi 编辑器由 Bill Joy 于 1976 年创建，并作为 1978 年中发布的 2BSD(第二版的伯克利 Unix)的一部分流传开来。最终，由于 **vi** 编辑器非常流行，因此 AT&T 公司在 System V 中也包含了这个编辑器，这使 **vi** 成为 Unix 编辑器的事实标准。多年以来，随着 **vi** 的维护责任由 Joy 传递给其他程序员，有许多人一直在开发 **vi**。经过如此众多的程序员对该程序的使用和修改，**vi** 得到了极大的增强。但是，直到 1992 年，所有的增强都相对微小。

这是因为 Joy 的原始设计非常出色，所以好长一段时间，根本就没有对 **vi** 进行较大改进迫切的需求。实际上，时至今日，当您使用 **vi** 时，看到的 **vi** 几乎与 BSD 用户在 20 世纪 70 年代末所看到的相同。这并不是说 **vi** 不能改进。只是这些改进是一些基本的改进，不至于使 **vi** 转变成一个显著不同的程序。

实际上，这种事情确实发生过。在 20 世纪 80 年代末，人们为非 Unix 系统创建了一个称为 STvi(通常写为 STevie)的开放源代码的克隆 **vi**。1988 年，一名荷兰程序员 Bram Moolenaar 接过 STvi，并使用它创建了一个新程序 Vim，该名称意味着“**vi** imitation, **vi** 仿制品”。在随后的几年内，Moolenaar 一直在开发 Vim，修复 bug，添加新特性，直到 1992 年，他才发行了该程序的第一个 Unix 版本。到现在为止，该程序进行了许多增强，因此 Moolenaar 改变了这个名称的含义。尽管该程序仍然命名为 Vim，但是 Moolenaar 声明，这一名称代表“**vi** improved, **vi** 改进版”。

在 20 世纪 90 年代，Vim 日益流行，特别是在 Linux 社区的那帮高手中，这些人包括程序员、系统管理员、网络管理员等。到 21 世纪 00 年代初期，Vim 已经非常流行，它成为大多数此类用户的编辑器首选，到 2005 年，许多 Linux 发行版都将 **vi** 替换为 Vim。实

际上，如果您现在使用的是 Linux，那么您的系统上极有可能只有 Vim，而没有 vi。如果是这种情况，那么当您输入 vi 命令，或者显示 vi 的说明书页(`man vi`)时，得到的将是 Vim，而不是 vi。大多数非 Linux 系统还不是这种情况。实际上，对于许多类型的 Unix 来说，可能系统上根本没有 Vim，除非自己安装了 Vim。

在本章中，我们不讨论 Vim，而是讨论标准的 vi，即许多年以来作为文本编辑主力的规范程序。我这样选择基于下述两点原因。

首先，Vim 其实并不是一种新版本的 vi，或者 vi 的扩展。Vim 是一种完全不同的程序，只是向后兼容 vi。这一区别非常重要。当运行 Vim 时，所使用的编辑器是一个拥有众多在 vi 中没有的复杂特性的编辑器。当然，因为 Vim 向后兼容，所以可以在 Vim 中使用 vi 的所有标准命令。但是，Vim 提供的新特性与 vi 相差甚大，从而使 Vim 的使用相对于 vi 的使用来说是一种完全不同的体验。

vi 对初学者来说已非常困难，而 Vim 更加困难，因为在学习 Vim 时，不仅要学习全部的 vi 命令和 ex 命令，而且还要学习 Vim 的全部附加命令。此外，在使用 Vim 的特殊特性时，解决问题的策略与使用 vi 时所用的完全不同。基于这一原因，如果您想学习 Vim，那么最好的策略是先学习 vi。

尽管这听起来有点难解，但实际上并不是这样。因为 Vim 非常制作得巧妙，所以可以像使用 vi 一样使用它。然后，一旦熟悉了 vi，就可以举一反三，学习如何充分利用 Vim 的功能。为了帮助大家的学习，在本章的末尾还将讨论一下 Vim 提供的额外特性，并示范如何着手学习这些特性。

我们准备只关注标准 vi 的第二个原因在于 vi 是标准。不管使用什么类型的 Unix 或 Linux，不管系统有多小，vi 极有可能是唯一可用的综合性文本编辑器。这样，即使您个人喜欢使用 Vim、Emacs、Nano、Pico 或其他编辑器，使用 vi 也是需要掌握的一个基本技能。

提示

如何知道系统使用的是 Vim，而不是 vi 呢？输入下述命令显示 vi 的说明书页：

```
man vi
```

如果看到的是 Vim 的说明书页，就可以知道系统使用的是 Vim 而不是 vi。

22.4 启动 vi

启动 vi 时，所使用的基本语法为：

```
vi [-rR] [file...]
```

其中 *file* 是希望编辑的文件的名称。

vi 程序非常复杂，因此可以想象出，它有许多选项。但是，大多数时候并不需要它们。实际上，在一般环境中，只需要知道两个选项，即 -r 和 -R，我们将在本章后面讨论这两个选项。

为了使用 **vi** 编辑一个已有的文件，只需指定文件的名称即可，例如：

```
vi essay
```

为了创建一个新文件，有两种选择。第一种选择就是指定一个文件名。如果这个文件不存在，**vi** 将会创建这个文件。例如，为了使用 **vi** 创建一个全新的文件 **message**，可以使用：

```
vi message
```

另外，还可以通过输入 **vi** 命令本身(不指定文件名)创建一个空文件：

```
vi
```

这会让 **vi** 创建一个没有文件名的新文件。然后，当保存数据时再指定文件名。

提示

如果忘记了正在编辑哪个文件(比想象的容易)，可以按 **^G***。这将显示文件的名称，以及您在这个文件中的位置。

22.5 启动 Vim: vim

正如前面所讨论的，在一些系统上，特别是 Linux 系统，**vi** 已经被 Vim 取代。如果您的系统上是这种情况，那么我的目标就是让您可以像使用 **vi** 一样使用 Vim。稍后，一旦习惯了 **vi**，您就可以自己学习如何使用 Vim 提供的扩展特性(我们将在本章末尾讨论 Vim 提供的扩展特性)。

现在，您只需知道如何将 Vim 作为 **vi** 来启动。一般而言，启动 Vim 就像启动 **vi** 一样。如果指定一个现有文件的名称，Vim 将打开这个文件。如果指定的文件不存在，Vim 将创建这个文件。如果没有指定文件名，Vim 将创建一个空文件，您可以在保存工作时再命名该文件。Vim 的基本语法为：

```
vim -C [-rR] [file...]
```

-r 和 **-R** 选项与 **vi** 相同，我们将在本章后面对此加以详细讨论。现在，我们讨论的选项是 **-C**。在 **vi** 和 Vim 中，有许多内部设置可以用来影响程序的行为。当以 **-C** 选项启动 Vim 时，它将改变 Vim 的设置，从而使 Vim 尽可能地像 **vi** 一样。当以这种方式使用 Vim 时，就称 Vim 运行在兼容模式(compatibility mode)**。除非您已经掌握了 **vi**，准备切换到完整的 Vim 中，否则最好总是以兼容模式启动 Vim。例如：

```
vim -C essay  
vim -C
```

* 正如第 7 章中讨论的，Unix 中约定将 **^**(音调符号)字符作为 **<Ctrl>** 的缩写。因此，**^G** 指的就是 **<Ctrl-G>**。

** 在一些系统上，当使用 **-C** 选项时，Vim 并不以兼容模式启动。如果您的系统正好是这种情况，那么您可以通过在 Vim 初始化文件(本章后面解释)中打开 **compatible** 选项强制 Vim 以兼容模式启动。

第一条命令告诉 Vim 希望处理的文件是 **essay**。如果这个文件存在，那么 Vim 就打开这个文件；如果这个文件不存在，那么 Vim 将创建这个文件。第二条命令告诉 Vim 创建一个全新的、没有命名的文件。当希望创建一个新文件，但是还没有想好名称时，可以采用这种方式。

如果您的系统上 Vim 已经取代了 **vi**，那么 **vi** 命令拥有和 Vim 命令相同的效果。在这些系统上，下述两条命令与前面的命令等价：

```
vi -C essay
vi -C
```

当没有指定文件名启动 Vim 时，程序将显示一些帮助信息(参见图 22-3)。该信息只是为了提供方便。一旦开始输入数据，Vim 将移除这些信息。

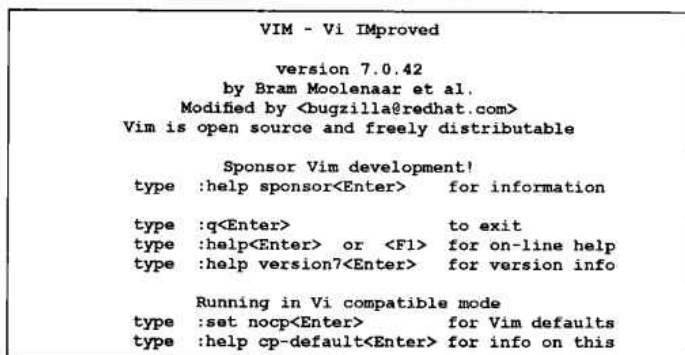


图 22-3 Vim 启动屏幕

当不指定文件名启动 Vim 时，将看到一个显示帮助信息的启动屏幕。该信息只是为了提供方便，一旦开始输入数据，这些信息就会消失。

提示

下面是一个简单的测试，可以查看系统是否使用 Vim 取代了 **vi**。输入没有文件名的 **vi** 命令：

```
vi
```

如果您看到一个几乎全空的屏幕，则使用的是标准 **vi**。如果您看到如图 22-3 所示的帮助信息，则使用的是 Vim(运行了这个测试之后，可以通过键入 **:q**，即一个冒号后面跟一个小写字母“q”来退出程序)。

提示

为了像 **vi** 一样使用 Vim，需要使用 **-C** 选项以兼容模式启动程序。为了方便起见，可以使用下述两条命令之一创建一个别名。其中，第一条命令针对的是 Bourne shell 家族(Bash、Korn Shell)，第二条命令针对的是 C-Shell 家族(Tcsh、C-Shell)：

```
alias vi="vim -C"
alias vi "vim -C"
```


为了使这个别名永久, 可以将该命令放在环境文件(参见第14章)中。一旦这样做了, 就可以使用 **vi** 命令以兼容模式运行 Vim, 而以 **Vim** 命令以固有模式运行 Vim。

22.6 命令模式和输入模式

在使用 **vi** 时, 存放数据的存储区域称为编辑缓冲区(editing buffer)。当告诉 **vi** 您希望编辑文件时, **vi** 就将文件的内容复制到编辑缓冲区中, 因此您处理的内容是数据的副本, 而不是原始数据。理解编辑缓冲区对 **vi** 的使用至关重要, 因此在后续阅读过程中一定要记住这一概念。

花点时间想一想使用字处理程序是什么样子的。在字处理程序中, 可以将光标移动到任何位置, 然后开始键入内容。当需要从文件的一个位置移到另一个位置时, 可以使用鼠标或者键盘上的特殊定位键。对于 PC 来说, 这些键包括 <PageUp>、<PageDown>、<Home>、<End> 以及光标控制(箭头)键。当需要使用命令时, 可以从下拉菜单或者使用特殊的键组合来选择某项。

1976 年, 当 Bill Joy 开发 **vi** 时, 终端还没有定位键。终端没有支持 GUI 的鼠标、下拉菜单、功能键, 甚至没有 <Alt> 键。终端上只有字母表中的字母、数字、标点符号和少数几个杂项键, 例如 <Shift>、<Ctrl>、<Return> 和 <Esc>。没有鼠标或定位键, 就无法找到一个简单的方法将光标从一个位置移动到另一个位置。没有下拉菜单或者特殊键, 用户指定命令(例如插入、修改、删除、复制或粘贴)时也就麻烦。

在设计 **vi** 时, Joy 选择的解决方法就是采用两种不同的模式。在命令模式(command mode)中, 所键入的键都被解释成命令。例如, 在命令模式中, 单个字母 **x** 就是删除一个字符的命令; 组合字母 **dd** 就是删除整行的命令。在命令模式中, 有许多 1 个字符和 2 个字符的命令, 而且为了掌握 **vi**, 必须学习这些命令。这听起来可能异常困难, 但是经过练习之后, **vi** 命令实际上相当容易使用。

第二种模式就是输入模式(input mode)。在这种模式中, 键入的任何内容都直接插入到编辑缓冲区中。例如, 在输入模式中, 如果键入 “Hello Harley”, 那么这 12 个字符将插入到编辑缓冲区中。如果按下 **x** 键, 则插入一个 “x”; 如果按下 **dd**, 则插入两个字符 “dd”。

该系统的可贵之处在于它不要求任何特殊的设备, 例如定位键或鼠标。这样, 就可以在任何类型的终端上使用 **vi**, 甚至是通过远程连接。该系统仅有的特殊键就是 <Ctrl> 和 <Esc>, 而这两个键在 20 世纪 70 年代末使用的每个普通终端上都有。

当然, 系统为了正常工作, 必须有一种方法从命令模式切换到输入模式, 并且还能在需要时切换回来。当 **vi** 启动时, 所处的模式是命令模式。在命令模式中, 可以使用几种命令(将在适当的时候学习)切换到输入模式。一旦处于输入模式中, 切换回命令模式就比较简单了, 只需按 <Esc>(Escape, 退出)键。如果已经处于命令模式中, 当按下 <Esc> 键时, **vi** 就会发出嘀嘀声。

如果您好奇为什么选择 <Esc> 完成这一任务, 则可以看一看图 22-4, 一张取自 ADM-3A 操作员手册的 ADM-3A 终端键盘布局图。正如前面所讨论的, 这就是 Bill Joy 在开发 **vi** 时使用的终端。注意 <Esc> 键的位置位于键盘的左边, 正好在 <Ctrl> 键的上方。对于这样重要的键来说, 这是一个绝佳的位置, 因为用左手的第 4 根或第 5 根手指可以方便地触及到这个键。



图 22-4 ADM-3A 终端的键盘布局

一张 Lear Siegler ADM-3A 终端的键盘布局图，取自 ADM-3A 操作员手册。这就是 Bill Joy 在开发 vi 文本编辑器时使用的终端类型。<Esc>键的绝佳位置正是 Joy 选择用它来从输入模式切换到命令模式的原因。详情请参见正文。

现在，请看一看您自己的键盘。注意<Esc>键已经移动到一个不怎么方便的位置上，即键盘的最左上角。<Esc>键的原来位置现在成为<Tab>键的位置，正好位于<CapsLock>键的上方。比较一下按<Tab>键和按<Esc>键之间的不同。当 Bill Joy 选择<Esc>键作为由输入模式改变到命令模式的功能键时，这个键非常容易触及，从而使模式的改变快速且简单。现在按<Esc>键需要抬起手去按，从而使这个过程变慢，模式的改变也没有那么方便了。这就是生活。

刚开始时，为了键入数据而不得不改变到一种特殊模式看上去好像很奇怪。不要担心。对于 vi 来说，熟不仅能生巧，而且还可以带来舒适安逸感，使您尽可能快地键入内容。如果您是按指法打字的人员，那么您将发现，一旦记住了基本的命令，vi 就非常易于使用，您可以手不离键盘完成任何事情。

为了使您对两种模式有所感觉，请考虑下述情形(现在还不必考虑细节，我们将在本章后面详细讨论)。您希望在文件 **schedule** 的中间添加一些数据。为了运行 vi，输入命令：

vi schedule

在启动过程中，vi 完成 3 件事情。首先，vi 将 **schedule** 文件的内容复制到编辑缓冲区。其次，vi 将光标定位到缓冲区第一行文本的开头。最后，vi 进入命令模式。

使用合适的命令将光标移动到希望添加新数据的位置上。然后键入一条命令，切换到输入模式，并开始键入内容。此时，所键入的任何内容都直接插入到编辑缓冲区中。当结束键入时，按<Esc>键切换回命令模式。然后，将编辑缓冲区中的内容保存到原始文件，并退出程序。

提示

注意，尽管是 vi 从一种模式改变到另一种模式，但是通常在谈话时会说是您(用户)自己进行了改变。例如，我可能说：“当您在命令模式中时，可以使用许多命令。”或者“为了向编辑缓冲区添加文本，您必须首先变到输入模式。”

对于计算机来说，这种谈话方式非常普通。这是因为敏感的人希望能控制他们的工具，而不愿意被动接受一切。

22.7 了解所处模式的方式

传统而言, **vi** 并不会告诉您现在处于什么模式, 而您却希望知道。我意识到这听起来相当混乱, 但是实际上并不是这样。一旦有经验之后, 您就会记住发生了什么事情, 因此大多数时候, 您知道自己处于什么模式之中。

如果您忘记了自己位于什么模式之中, 那么记住这一点: 如果位于命令模式之中, 那么按下<Esc>键, **vi** 就会发出嘀嘀声。因此, 如果不确定位于哪一种模式之中, 只需按<Esc>键两次。这可以确保您位于命令模式之中, 并至少发出一次嘀嘀声(如果您在输入模式中的话, 那么第一次按<Esc>键将改变到命令模式, 第二次按<Esc>键才会发出嘀嘀声。如果您已经在命令模式中, 那么两次按<Esc>键都将发出嘀嘀声)。

然而, 您可能会问, 难道 **vi** 就不能显示处在什么模式之中吗? 实际上, 一些版本的 **vi** 也提供了显示模式的方法, 即设置一个内部选项 **showmode**(我们将在本章后面讨论 **vi** 的选项)。使用的命令是:

```
:set showmode
```

一旦设置了这个选项, **vi** 就会在屏幕的底部显示一个消息以指示当前的模式(实际消息可能会根据 **vi** 版本的不同而有所不同, 但是都不难理解, 可以很容易看出处于什么模式之中)。如果您决定总是设置 **showmode** 选项, 那么您可以将这条命令放在 **vi** 的初始化文件中, 这样无论何时启动 **vi**, 它都会自动地设置这个选项(我们将在本章后面讨论初始化文件)。

如果您是一名 **Vim** 用户, 则不必设置这个选项。默认情况下, 无论何时, 当进入输入模式时, **Vim** 将在屏幕的左下角显示如下提醒:

```
-- INSERT --
```

虽说可以看到一个可视的提醒是一件好事, 但是事实是这并不必要。正如前面所述, 一旦习惯了 **vi**, 并拥有了命令模式和输入模式之间来回切换的经验之后, 就总是能够知道自己处于什么模式之中。基于这一原因, 许多有经验的 **vi** 用户不愿意设置 **showmode** 选项, 即使系统提供了这个选项。他们真的不需要这个选项——其实经过一段时间练习之后, 您也将不需要这个选项。

提示

不管您有多聪明, **vi** 会使您更聪明。

22.8 以只读方式启动 vi: view、vi -R

有时候, 需要使用 **vi** 查看一个重要的文件, 但又不希望改变这个文件。有两种方法可以实现这一目的。第一种方法, 可以使用 **-R(read-only, 只读)** 选项启动 **vi** 程序。这将告诉 **vi**, 您不希望将数据保存回原始文件(这个选项适用于 **vi** 和 **Vim**)。第二种方法, 可以使用

view 命令启动该程序。

vi -R 和 **view** 之间其实没有什么区别, 哪个好记就可以使用哪一个。因此, 下面两条命令是等价的:

```
vi -R importantfile
view importantfile
```

两条命令都使用文件 **importantfile** 以只读方式启动 **vi**。以这种方式使用 **vi** 可以保护重要的数据, 以免其被不小心修改。

您可能奇怪, 为什么有人希望使用 **vi** 来处理一个不能改变的文件呢? 如果只希望显示文件, 为什么不使用 **less**(参见第 21 章)呢? 答案是 **vi** 的功能非常强大, 因此在显示文件时, 许多人愿意使用 **vi** 命令, 而不是 **less**。一旦掌握了 **vi** 的使用方法, 就会体会到这种感觉, 特别是当需要查看复杂的大型文件时。

22.9 系统失败后数据的恢复

在编辑文件的过程中, 有时候可能会遇到系统关机, 或者失去了与系统的连接。如果是这样, 那么 **vi** 通常可以将数据恢复出来。

前面讲过, 当使用 **vi** 时, 正在编辑的数据保存在编辑缓冲区内。**vi** 编辑器会时不时地将编辑缓冲区中的内容保存到一个临时文件中。通常, **vi** 在编辑完成时将删除该文件。但是, 如果程序非正常终止, 那么这个临时文件还会存在, 因此我们可以使用这个文件来恢复数据。

为了恢复数据, 需要使用 **-r(recover, 恢复)** 选项启动 **vi**:

```
vi -r
```

这将显示所有可以用来恢复数据的文件。接下来就可以使用 **-r** 选项, 后面跟希望恢复的文件的名称重新启动 **vi**。例如:

```
vi -r test.c
```

这将恢复文件, 恢复到系统关机时文件的内容。

注意: 一定要小心, 不要将 **-r(recover, 恢复)** 选项与 **-R(read-only, 只读)** 选项混淆。

提示

Vim 通过将编辑缓冲区保存在一个交换文件(swap file)中, 提供了出色的恢复功能。交换文件与正在编辑的文件存储在同一个目录中(我们将在第 23 章和第 24 章中讨论目录)。每当键入 200 个字符之后, 或者有 4 秒没有键入内容时, 交换文件都会自动地更新。

为了恢复文件, 必须使用 **rm** 命令(参见第 25 章)删除交换文件, Vim 不会自动完成这一操作。

交换文件的名称包含一个.(点号), 后面跟着原始文件的名称, 再后面跟着 **.swp**。例如, 如果正在编辑程序 **test.c**, 那么交换文件将是 **.test.c.swp**。如果不删除交换文件, 那么下一

次编辑原始文件时，Vim 将以一个稍微有所不同的名称创建一个新交换文件，例如 `test.c.swp`。

22.10 停止 vi

vi 的停止方式有两种。大多数时候，用户都希望保存完工作后退出 **vi**。但是，如果由于不小心而搞乱了数据，则可能希望不保存数据而退出程序，以保持文件的原始内容不变。不管是哪一种情况，都必须在命令模式下输入一条退出命令。如果位于输入模式中，则必须首先按 `<Esc>` 键切换到命令模式。

在需要保存工作然后再退出时，使用的命令是 **ZZ**(稍后再解释这个名称的含义)。即按下并保持 `<Shift>` 键，再按 `<Z>` 键两次。该命令不需要按 `<Return>` 键：

ZZ

如果希望不保存工作而退出程序，则可以使用命令 **:q!**。在键入这条命令之后，需要按 `<Return>` 键：

:q!<Return>

在使用 **:q!** 命令之前，要好好地斟酌一番。一旦没有保存数据而退出程序，就没有办法恢复数据了。

在本章后面，我将解释为什么第二条命令以一个冒号开头，以及为什么需要按 `<Return>` 键。放心，它拥有明确的含义(**ZZ** 是 **vi** 命令，而 **:q!** 是 **ex** 命令)。现在，我所提及的内容就是在 Unix 中，**!** (感叹号)字符有时候用来表示希望忽略某些类型的自动检查。在 **:q!** 中，**!** 告诉 **vi** 不要检查是否保存了数据。

提示

当使用 Vim 时，如果不小心输错了 **:q!** 命令，则可能陷入很大的麻烦。原因在于，对于 Vim 来说，键入 **q** 是记录宏的信号(我们将在本章后面讨论宏)。如果遇到了这种情况，您将在窗口底部看到一个消息“recording”。这时不要惊慌，为了停止宏记录功能，只需再次键入 **q**(代表退出)，并等待这个消息消失。

名称含义

ZZ

使用一个快捷的方式保存工作并退出 **vi** 确实有意义，但是为什么是 **ZZ** 呢？

假设该命令拥有一个比较简单的名称，例如 **s**(代表 save，即保存)。这将比较方便，但是如果您认为自己处于命令模式中，但其实您处于输入模式中，于是您键入了 **s**，这时会发生什么情况呢？您准备键入数据，但是在不知道的情况下，您已经键入了一个“**s**”，将程序停止。

选择名称 **ZZ** 是因为，尽管它也容易键入，但是一般不会由于不小心而键入它。

22.11 vi 使用屏幕的方式

此时，我希望花点时间讨论几个与 **vi** 使用屏幕相关的小话题。屏幕底部的一行称为**命令行**。**vi** 以两种不同的方式使用这一行：显示消息以及显示键入命令时的命令(参见下一节)。屏幕上的其他行都用于显示数据。

如果编辑缓冲区中只包含少量的数据，那么这些数据可能无法铺满整个屏幕。例如，假设终端或者窗口包含 25 行。屏幕最底部的一行是命令行，这样只剩下 24 行显示数据。假设编辑缓冲区只包含 10 行数据。如果 `vi` 将其余 14 行显示为空白行，则有可能出现混乱。毕竟，用户也确实有可能将空白行作为数据的一部分。

为此，vi 以一个~字符作为每个空行的开头。图 22-5 中示范了这样一个例子。在向编辑缓冲区中添加新行时，它们将占用越来越多的屏幕空间，最终使~消失。

[illegible]

图 22-5 vi 显示空行的方式

屏幕最底部的一行是命令行，vi 以两种方式使用这一行：显示消息以及显示键入的命令。其他行都用于显示数据。

当编辑缓冲区包含的数据不足以占据所有行时，vi 通过显示~字符标识空行。在这个例子中，总共有 17 个空行，每个空行都通过显示一个~字符标记。随着插入到编辑缓冲区中的数据行的增多，空行将被逐渐使用，~字符将逐渐消失。

大多数时候，用 **vi** 编辑的数据只包含纯文本：字符、字母、数字、标点符号等。但是，如果需要，也可以在编辑缓冲区中插入控制字符(参见第 7 章)。为了这样做，需要按 **^V** 键，后面跟希望输入的控制字符。例如，如果希望键入一个实际的 **^C** 字符，则需要按 **^V^C**。如果希望输入一个 **^V**，则需要键入 **^V^V**。

当 **vi** 显示控制字符时，将看到一个[^]字符，后面跟一个字母，例如[^]C。记住，它实际上是一个单独的字符，虽然它在屏幕上占两个字符的空间。

正如第 18 章中解释的，制表符字符是[^]I。跟其他 Unix 程序一样，**vi** 编辑器也假定制表符设置为每 8 个位置一个(此值可以改变，但是大多数人不愿意这么做)。在向编辑缓冲区中插入一个制表符时，**vi** 不显示[^]I，而是显示多个空格，使数据看上去是根据制表符定位的。这只是为了提供方便，这些额外的空格其实是不存在的。实际上，每个制表符只是一个单独的字符([^]I)。

最后，在屏幕变得错乱时(例如，如果您远程工作，线路上有干扰)，可以通过按[^]L 键告诉 **vi** 在屏幕上重新显示各行。

22.12 使用 vi 和 ex 命令

前面已经解释过 **vi** 和 **ex** 其实是同一个程序的两种不同表现形式。这就意味着在使用 **vi** 时，可以同时使用 **vi** 命令和 **ex** 命令。

大多数 **vi** 命令都是单字母或双字母的形式。例如，将光标向前移动一个单词，可以使用 **w** 命令(只需在命令模式下键入“w”)。为了删除当前行，可以使用 **dd** 命令(只需键入“dd”)。因为 **vi** 命令如此之短，所以键入时命令不回显。

大多数 **vi** 命令都无需按<Return>键。例如，一旦键入“w”，光标就立即向前移动一个单词的位置。一旦键入“dd”，当前行就立即消失。如果不小心误输了不正确的 **vi** 命令，则会听到嘀嘀声。但是，不会显示错误消息(这是什么观点?)

ex 命令要比 **vi** 命令长一些且更复杂。基于这一原因，在键入过程中它们会回显在命令行上。所有的 **ex** 命令都以一个:(冒号)开头。例如，下述命令删除 1 至 5 行：

```
:1,5d
```

下述命令将所有的“harley”修改成“Harley”(现在还不用关心细节问题)。

```
:%s/harley/Harley/g
```

一旦键入了打头的冒号，**vi** 就将光标移动到命令行上(屏幕的最底部)。在键入命令时，每个字符都将回显。当结束键入时，必须按下<Return>键。

如果在按下<Return>键之前，发现了错误，则有两种选择：第一，按下<Esc>键，彻底取消这个命令；第二，使用图 22-6 所示的特殊键(详情请参见第 7 章)对命令进行纠正。另外，也可以在输入模式下使用这些键在键入过程中进行纠正。

键	作用
<Backspace>/<Delete>	删除键入的最后一个字符
[^] W	删除键入的最后一个单词
[^] X/ [^] U	删除整行

图 22-6 在使用 vi 的过程中用来进行纠正的键

当在 **vi** 编辑器中进行键入时，有 3 个标准的 Unix 键可以用来进行纠正。详情请参见第 7 章。

在一些系统上，当进行纠正时，光标虽然回移，但是字符并不从屏幕上消失。例如，假设您输入了：

```
:1,5del
```

在按<Return>键之前，您意识到命令尾部无需键入 `el`。因此，您按下<Backspace>键两次。光标将向后移动两个位置，但是您仍然可以看到这两个字符。不要担心，这两个字符已经不存在了。现在可以放心地按<Return>键了。

22.13 学习 vi 命令的策略

vi 编辑器拥有众多的命令。为了方便起见，我们可以按如下方式对这些命令进行分类：

- 移动光标的命令
- 进入输入模式的命令
- 进行修改的命令

我的目标就是让您在每一组命令中学习足够多的命令，从而能够随时归纳出执行任何编辑任务所需的策略。下面举一个例子，示范上面所说的意思。在您工作时，光标显示您在编辑缓冲区的当前位置。为了在缓冲区中插入新的数据，需要采用如下策略：

- (1) 确保处于命令模式中。
- (2) 将光标移动到希望插入数据的位置上。
- (3) 切换到输入模式。
- (4) 输入数据。
- (5) 按<Esc>键切换回命令模式。

一旦学会了基本的 **vi** 命令，您就会发现实现任何特定的策略都有许多种方式。选择哪一种方式取决于特定的场合以及您的技能水平。

您可能对 **vi** 提供有如此之多的命令感到惊奇。例如，进入输入模式有 12 种不同的命令；在命令模式中，有 40 种不同的命令移动光标(这些只是简单的光标命令)。

可以想象，没有人需要知道 12 种进入输入模式或者 40 种移动光标的方式。但是，我希望您学习尽可能多的命令，因为——不管您相信还是不相信——这会使 **vi** 更易于使用。

例如，假设您希望从屏幕的左上角移动到屏幕中间离右边缘几个单词的地方。您可以每次一个位置地移动光标，但是这种方法很慢，而且比较笨拙。如果您知道 40 种移动光标的命令，则可以从中学选取最适用的，也许只需按三、四个键就可以立即移动到希望的准确位置上去。

在本章中，我们只讨论基本的 **vi** 命令和 **ex** 命令。更全面的信息请参见附录 C，该附录中包含所有重要的命令。我的建议就是不断地自学，直至了解附录 C 中的全部命令。时不时地花一些时间自学一条新命令，然后再练习这条新命令，这是很有益的。所有的命令都是有用的，都是值得练习的。在阅读本章剩余部分时，希望您能坐在计算机前，遵循书中的指示练习各条命令。在讨论每条新命令时，希望您能够体验一下它们。

提示

vi 的使用艺术在于能够选取最佳的命令，尽可能简单快速地执行任务。

22.14 创建一个练习文件

在阅读本章内容时，需要使用一个文本文件来练习编辑。使用下述两条命令之一就可以创建一个这样的文件：

```
cp /etc/passwd temp
man vi > temp
```

第一条命令通过复制系统的口令文件创建一个小文件。第二条命令通过复制 vi 的说明书页创建一个大文件(cp 命令在第 25 章解释，口令文件在第 11 章解释，联机手册在第 9 章解释，而使用>进行标准输出重定向在第 15 章解释)。

这两条命令都将创建一个 **temp** 文件，可以使用这个文件进行练习。一旦拥有了这样一个文件，就可以输入下述命令编辑这个文件：

```
vi temp
```

在完成编辑之后，可以使用下述命令移除(删除)这个文件：

```
rm temp
```

(rm 命令在第 25 章解释。)

22.15 移动光标

在任何时候，光标必然位于屏幕的某一行上。这一行就称为当前行。在当前行中，光标将位于某个特定字符之上或者之下，这个字符就称为当前字符。许多 vi 命令对当前行或者当前字符执行操作。例如，**x** 命令删除当前字符，**dd** 命令删除当前行。

无论何时，当移动光标时，新位置的字符就成为当前字符。同样，如果将光标移动到一个新行上，那么这一行就成为当前行。每当将光标移动到一个当前不在屏幕上的行上时，vi 将显示编辑缓冲区不同部分，从而使新行可见。换句话说，就是如果要从编辑缓冲区的一部分跳到另一个部分，只需简单地移动光标。

在 vi 中，有许多不同的命令可以移动光标，这意味着有许多不同的方式可以从编辑缓冲区的一个位置跳到另一个位置。我的目标就是讲授这些命令中的大多数。然后，每当需要从编辑缓冲区的一个位置跳到另一个位置上去时，您能够发现哪个命令序列最佳。不久以后，选取最快的光标移动命令将成为您的第二天性。

在一些情况中，有多种方式可以完成完全相同的光标移动。例如，将光标向左移动一个位置就有 3 种不同的命令。在这些情况下，没有必要学习所有等价的命令。只需挑选一

个自己最喜欢的命令，并进行练习即可。下面开始讨论其他的命令。

将光标移动一个位置，有许多方法可以选择。最好是使用 **h**、**j**、**k** 和 **l** 命令，它们的作用如下所示：

- h** 将光标向左移动一个位置
- j** 将光标向下移动一个位置
- k** 将光标向上移动一个位置
- l** 将光标向右移动一个位置

为什么会选择这几个键呢？原因有两点。首先，如果您是按指法打字的人，那么这 4 个键很容易用右手的手指进行敲击，因此能够非常方便地移动光标(看看自己的键盘)。

其次，正如本章前面讨论的，**vi** 是在 1976 年由 Bill Joy 使用 ADM-3A 终端开发的。图 22-7 给出了 ADM-3A 键盘的特写，展现了这 4 个键。注意字母上的箭头，在设计 ADM-3A 时，就是使用这 4 个键作为光标控制键的，因此 Joy 以相同的方式使用它们也很自然。

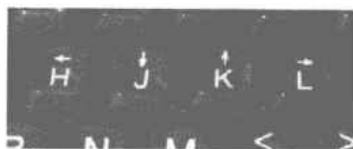


图 22-7 ADM-3A 终端上的 H、J、K 和 L 键

当 Bill Joy 在 1976 年开发 **vi** 时，他使用的是一台 Lear Siegler ADM-3A 终端。对于这个终端来说，H、J、K 和 L 键不仅仅用来键入字符，而且还用来控制光标(注意箭头)。基于这一原因，Joy 选择使用相同的 4 个键在 **vi** 中移动光标。详情请参见正文。

如果您是按指法打字的人，那么使用 H、J、K 和 L 键移动光标特别方便。此外，还有其他键可以用来控制光标，而且更容易记忆。如果您的键盘上有光标控制键(箭头键)，那么您可以使用它们(我将称它们为<Left>、<Down>、<Up>和<Right>)。另外还可以使用<Backspace>键向左移动一个位置，或者使用<Space>键向右移动一个位置。

- <Left> 将光标向左移动一个位置
- <Down> 将光标向下移动一个位置
- <Up> 将光标向上移动一个位置
- <Right> 将光标向右移动一个位置
- <Backspace> 将光标向左移动一个位置
- <Space> 将光标向右移动一个位置

另一种上下移动光标的方法就是使用-(减号)和+(加号)命令。按-键将把光标移动到上一行的开头，按+键将把光标移动到下一行的开头。另外还有一种方法，按<Return>键也可以移动到下一行的开头。

- 将光标移动到上一行的开头
- + 将光标移动到下一行的开头
- <Return> 将光标移动到下一行的开头

在当前行中，**0**(数字 0)命令将光标移动到行的开头；**\$**(美元符号)命令将光标移动到行的末尾。如果当前行是缩进的，则可以使用[^](音调符号)将光标移动到本行不是空格或者制表符的第一个字符上去。

- 0** 将光标移动到当前行的开头
- \$** 将光标移动到当前行的末尾
- [^] 将光标移动到当前行的第一个非空格/制表符的字符上

除了按字符或按行移动光标之外，还有几条命令可以用来以单词为单位移动光标(在 vi 中，单词就是一串字母、数字或下划线字符)。为了向前移动，可以使用 **w** 或者 **e** 命令。**w** 命令将光标移动到下一个单词的第一个字符上，**e** 命令将光标移动到下一个单词的最后一个字符上。为了向后移动，可以使用 **b** 命令将光标移动到上一个单词的第一个字符上。使用 **w**、**e** 或 **b**，可以快速准确地将光标移动到期望位置上，节省大量的按键动作。

- w** 将光标移动到下一个单词的词首
- e** 将光标移动到下一个单词的词尾
- b** 将光标移动到上一个单词的词首

所有这 3 条命令都在每个标点符号字符处停止，如果数据中不包含大多这样的字符，这几条命令就比较合适。但是，如果数据中有大量的标点符号，那么以这种方式移动光标就比较慢。作为替代，可以使用 **W**、**E** 和 **B** 命令。这几条命令的运行方式与前 3 条命令相同，只是它们仅识别作为单词末尾的空格和新行字符。

- W** 同 **w**，忽略标点符号
- E** 同 **e**，忽略标点符号
- B** 同 **b**，忽略标点符号

例如，假设光标现在位于下述行的开头：

```
This is an (important) test; don't forget to study.
```

如果连续按 **w** 键，则光标会停在每个圆括号、分号、撇号(单引号)以及每个单词的开头。也就是说，要到达这一行的最后一个单词需要按 13 次 **w** 键。如果使用 **W**，光标将只在每个空格处停止。要到达该行的最后一个单词只需按 8 次 **W** 键(最好现在就试一试这些命令)。

对于更大范围的移动，可以使用圆括号命令从一个句子跳到另一个句子：

-) 向前移动到下一个句子
- (向后移动到上一个句子

同样，花括号命令可以从一个段落跳到另一个段落：

- { 向前移动到下一个段落
- } 向后移动到上一个段落

再一次强调，这些命令应该自己试一试，以确保正确理解它们的作用。在练习每条命令时，要注意到能够以单词、句子和段落为单位跳动是多么的便利。然后再花点时间考虑下述问题：英语散文由单词、句子和段落构成，除 **vi** 之外，为什么没有其他文本编辑器和字处理程序允许直接以单词、句子和段落为对象进行处理呢？

在 **vi** 中，句子的正式定义是一个字符串，以句号、逗号、问号或感叹号结尾，后面至少跟两个空格或一个换行符(换行符标记每行的末尾，参见第 7 章)。

换句话说，**vi** 所识别的句子，必须保证后面是两个空格，或者是行尾。之所以这样要求，是因为两个空格可以使 **vi** 分辨出是一个句子还是一个单词(单词之间以一个空格分隔)。考虑下述例子，其中包含两个句子，用一个句号和两个空格分隔开：

Meet me at the Shell Tower at 6pm. Is this okay with you?

段落定义为以一个空白行开头并以一个空白行结束的一块文本区域。换句话说，在行首放一个制表符来标识一个新段落是远远不够的。

提示

通常，在两个句子之间键入两个空格，在两个段落之间放入一个空白行，是一个很好的习惯。这样做有 3 个好处：

第一，它使文章更易于阅读。

第二，在使用 **vi** 时，可以方便地从一个句子移动到另一个句子(使用(和)键)，或者从一个段落移动到另一个段落(使用{和}键)。

第三，如果将文本发送给希望编辑这个文本的人，那么他会感到操作起来容易得多。例如，假设您通过电子邮件给某人发送一条消息，在这个人回复时，他会很有礼貌地对您的消息进行编辑。如果原始消息的排版格式很好，那么他就可以更容易地删除整个句子或者段落。这一点看起来似乎微不足道，但事实上并非如此。

预言：当您习惯了使用 **vi** 之后，就会发现如果有人在纯文本中的两个句子之间只放一个空格会让您很恼火。

有时候，可能需要从屏幕的某一部分大范围地移动到另一部分。为了进行这样的移动，可以使用 **H**、**M** 或 **L** 命令。这些命令可以分别跳到屏幕的顶部、中间或者底部(可以联想一下“high”、“middle”和“low”这几个单词)。

H 将光标移动到屏幕的顶部

M 将光标移动到屏幕的中间

L 将光标移动到屏幕的最后一行

通常，移动光标的艺术在于使用尽可能少的击键动作到达希望的位置。下面举例说明。假设光标处于屏幕的顶部，屏幕上的最后一行数据为：

today if you can. Otherwise give me a call.

您希望将光标移动到“call”这个单词的“c”字母上，从而可以插入单词“phone”。

您可以连续按<Down>键移动到希望的那一行上，然后按<Right>键多次，移动到希望

的单词。但是，用另一种方法只需 3 次击键就可以完成整件事情：

L\$b

- (1) 将光标移动到屏幕的最后一行(L)。
- (2) 将光标移动到这一行的末尾(\$)。
- (3) 将光标移动到上一单词的开头(b)。

为了增强光标移动命令的功能，可以通过在命令前面键入一个数字来重复执行光标移动命令，这个数字称为**重复次数(repeat count)**。例如，为了向前移动 10 个单词，可以键入：

10w

注意数字之后不能有空格。下面再示范两个例子。为了向下移动 50 行，可键入下面任意一条命令：

50j

50<Down>

50+

50<Return>

为了向后移动 3 个段落，可以对{(左花括号)命令使用一个重复次数：

3{

通常，对任何 vi 命令使用重复次数都是允许的，只要这样做有意义。

提示

每当需要从一个位置移动到另一个位置时，应该尽可能设法使用最少的击键完成光标的正确定位*。

22.16 在编辑缓冲区中移动

无论什么时候，vi 都会尽可能地将编辑缓冲区中的内容铺满屏幕。当处理的文件包含大量的文本时，一次只能看到一部分文件。例如，如果终端或窗口只有 25 行，那么 vi 每次只能显示 24 行(记住，vi 不在屏幕的最底部一行显示文本，这一行用于显示命令行)。当希望查看另一部分文本时，需要将光标移动到编辑缓冲区中的对应位置上。完成这种移动的命令有好几种。

首先，可以使用[^]F(forward, 向前)命令移动到下一屏(记住，[^]F 指的是<Ctrl-F>)。相反的命令是[^]B(backward, 向后)，该命令移动到上一屏。另外还有两种命令：[^]D 下移半屏，

* 如果您是一名学生，那么这种方法有助于让您的学生生活更丰富多彩。您可以带上自己的膝上型电脑去学生中心，与他人打赌，说您移动 vi 光标的速度比任何人都要快。开始时，您可以使用一些像<Up>和<Down>这样小范围移动的键。在输了几次，参赛者增多之后，就可以使用 H、M 和 L 命令，接着使用句子和单词的重复命令，将所有人击败。

^U 上移半屏。当希望在文件中快速移动时,可以使用**^F**和**^B**。当希望移动的范围较小时,可以使用**^D**和**^U**。

^F 向下(前)移动一屏

^B 向上(后)移动一屏

^D 向下移动半屏

^U 向上移动半屏

通常,如果在光标移动命令之前键入一个数字,那么这个数字就是重复次数。例如,为了一次向下移动 6 屏,可以键入:

6^F

为了一次向上移动 10 屏,可以使用:

10^B

因为可以使用**^F**和**^B**命令以这种方式跳转一段很长的距离,所以没有必要再对**^D**和**^U**命令使用重复次数。因此,当在**^D**或**^U**命令前面键入数字时,该数字拥有完全不同的含义:设置这两条命令应该跳转的行数。例如,考虑下述命令:

10^D

10^U

这两条命令都告诉 **vi** 跳转 10 行,并且所有随后的**^D**和**^U**命令也都跳转 10 行(直到重设重复次数复位)。如果您喜欢,还可以将行数设置为一个较大的值。例如,如果希望一次跳转 100 行,可以使用:

100^D

100^U

除非改变这个数字,否则接下来所有的**^D**和**^U**命令都将跳转 100 行。

22.17 跳转到前一位置

有很多时候,当把光标移动了很长一段距离之后,发现又希望将其移回去。有时候,这种移动是有意的。例如,您可能跳转到编辑缓冲区的末尾,添加一行,然后又希望返回到原来那一行。这种情形也有可能是无意间发生的,例如当发现错误时,已经离错误位置好长一段距离了。

在这些情况下,可以使用````命令返回到前一位置(也就是说,连续键入两个反引号)。为了测试这条命令,用一个大文件启动 **vi**,然后使用带重复次数的 **G** 命令跳转到第 10 行:

10G

现在使用 **I**(小写字母“i”)命令将光标移动到这一行的第 8 个字符:

```
81
```

接下来,使用 **G** 命令跳转到编辑缓冲区的末尾,即键入:

```
G
```

为了返回到前一位置(第 10 行的第 8 个字符),可以连续键入两个反引号:

```
``
```

该命令的一个变体就是使用"(两个单引号)取代两个反引号。这将跳转到行的开头,而不是在行的中间。为了测试这个命令,我们再次将光标移动到第 10 行的第 8 个位置上,然后跳转到文件的末尾:

```
10G
```

```
81
```

```
G
```

现在,连续键入两个单引号:

```
''
```

注意光标现在位于第 10 行的开头。

该命令还有一个更强大的版本,允许用不可见的名称标识任意行。然后,每当需要时就可以使用这个名称跳转到这一行。为了以这种方式标识一行,可以键入 **m**,后面跟一个字母。这个字母就是这一行的名称。例如,为了用名称“a”标记当前行,可以键入:

```
ma
```

为了跳转到一个标记行,可以键入一个` (反引号)或一个' (单引号),后面跟这一行的名称,例如:

```
`a
```

```
'a
```

第一条命令(反引号)跳转到标记行的准确位置(即标记这一行时的位置)。第二条命令(单引号)只跳转到标记行的开头。

提示

当标记行时,可以使用任何字母。原则上,这允许标记 26 行(从 **a** 到 **z**)。实践中,极少需要同时标记多于两行。标记一行时,最简单的方法是键入 **mm**,然后就可以通过键入 **'m** 跳转到这一行(如果标记两行的话,可以使用 **ma** 和 **mm**)。

一旦习惯了键入这些组合,就会很自然地标记某一行,将光标移动到其他地方做些事情,然后再跳转回原始行,而且整个过程中不必将手离开键盘(好好体验一下 **vi** 的强大功能)。

22.18 搜索模式

另一种在编辑缓冲区中来回移动的方法就是跳转到包含某一特定模式的行上，这时需要使用/(斜线)和?(问号)命令。

一旦按下了/键，**vi** 就会在命令行上(屏幕的底部)显示一个/字符。然后就可以键入希望的模式并按<Return>键。这将告诉 **vi** 搜索该模式的下一个匹配。如果希望再次搜索同一个模式，可以再次键入/并按<Return>键。

下面举例说明。您正在编辑一个人员列表，要给他们发钱，您希望查找模式“Harley”的下一个匹配项。可以键入：

```
/Harley
```

现在按下<Return>键，光标将跳转到接下来包含该模式的行上。为了重复搜索该模式并再次跳转，可以只键入/本身，然后按<Return>键：

```
/
```

因为没有指定新模式，所以 **vi** 假定和上一次的/命令使用相同的模式。默认情况下，**vi** 的搜索区分大小写。因此，下述两条命令是不相同的：

```
/Harley
```

```
/harley
```

当 **vi** 查找模式时，它从光标位置开始向前搜索。如果光标到达编辑缓冲区的末尾，**vi** 就会转过头来从头开始搜索。通过这种方式，**vi** 可以搜索整个编辑缓冲区，不管起始位置位于何处。

为了向后搜索，可以使用?命令。例如：

```
?Harley
```

这与/的作用相同，只是 **vi** 向后搜索。一旦使用?指定了一种模式，就可以再次使用?本身向后搜索同一模式：

```
?
```

如果 **vi** 到达编辑缓冲区的开头，**vi** 将返回到文件的末尾并继续向后搜索。通过这种方式，**vi** 可以向后搜索整个编辑缓冲区。

一旦使用/或?指定了模式，就可以用两种便捷的方式继续以原有模式进行搜索。**n(next)**，下一个)命令采取和原始命令相同的方向进行搜索。**N**(大写字母“N”)命令以相反方向进行搜索。例如，假设您之前输入了下述命令：

```
/Harley
```

现在您希望查找同一模式的下一次出现，那么所需做的就是按 **n** 键(不用按<Return>键)。其作用与前面输入的不带模式的/<Return>一样。要重复地对同一模式进行搜索，可以

重复地按 **n** 键。如果按 **N** 键，**vi** 将进行反向搜索。和其他搜索命令相同，在需要时，**n** 和 **N** 都会在编辑缓冲区的末尾(或开头)跳转。

n 和 **N** 的准确含义取决于原始的搜索方向。例如，假设您输入一个向后搜索的命令：

?Harley

那么按 **n** 键将向后搜索(相同方向)，按 **N** 键将向前搜索(相反方向)。

提示
尽管 **/** 和 **?** 搜索命令是为 **vi** 开发的，但是其他程序也使用它们。例如，在用 **less** 程序(参见第 21 章)显示文件时，就可以使用完全相同的命令。

为了获得更大的灵活性，可以使用正则表达式指定模式(正则表达式在第 20 章中详细讨论过，该章示范了许多例子)。出于参考目的，图 22-8 示范了在正则表达式中拥有特殊含义的各种元字符。

元字符	含义
.	匹配除新行字符之外的任意单个字符
^	锚：匹配行的开头
\$	锚：匹配行的末尾
\<	锚：匹配单词的开头
\>	锚：匹配单词的末尾
[list]	字符类：匹配 list 中的任何字符
[^list]	字符类：匹配不在 list 中的任何字符
\	引用：从字面上解析元字符

图 22-8 vi 搜索使用的正则表达式元字符

在 **vi** 中使用 **/** 和 **?** 命令搜索时，可以通过为搜索模式使用正则表达式增强搜索的能力。出于参考目的，这里示范了在正则表达式中最常使用的元字符。详情请参见第 20 章。

下面举几个示范正则表达式能力的例子。为了搜索一个以“H”开头，后面跟任意两个字符的模式，可以使用：

/H..

为了搜索一个以“H”开头，后面跟任意两个小写字母的模式，可以使用：

/H[a-z][a-z]

为了搜索一个以“H”开头，后面跟 0 个或多个小写字母，再后跟一个“y”的模式，可以使用：

/H[a-z]*y

为了搜索下一个以“Harley”开头的行，可以使用：

`/^Harley`

下面进行总结:

`/regex` 向前搜索指定的正则表达式
`/` 向前重复搜索前一正则表达式
`?regex` 向后搜索指定的正则表达式
`?` 向后重复搜索前一正则表达式
`n` 重复上一条/`或?`命令, 搜索方向相同
`N` 重复上一条/`或?`命令, 搜索方向相反

22.19 使用行号

从内部讲, **vi** 通过给编辑缓冲区中的每一行编一个号来区分各行。如果想查看这些行号, 则可以打开 **number** 选项(我们将在本章后面讨论 **vi** 选项)。所使用的命令是:

```
:set number
```

例如, 假设您使用 **vi** 写一篇应用哲学论文。编辑缓冲区中包含:

```
I have a little shadow that goes  
in and out with me,  
And what can be the use of him  
is more than I can see.
```

如果输入命令:**set number**, 将会看到:

```
1 I have a little shadow that goes  
2 in and out with me,  
3 And what can be the use of him  
4 is more than I can see.
```

意识到下述一点非常重要, 即行号并不是数据的真正组成部分。它们只是为了使用方便而设置的。如果想除去行号, 可以按如下方式关闭 **number** 选项:

```
:set nonumber
```

如果行号关闭, 则可以通过按 **^G** 键查看位于文件何处。这将显示文件的名称, 以及光标在文件中的位置。

行号有两个主要用途。第一, 从后面可知, 可以在许多 **ex** 命令中使用行号。第二, 可以使用 **G**(go to, 跳到)命令跳转到指定行。只需简单地键入行号, 后面跟上 **G** 即可。注意不要键入空格, 也不需要按 **<Return>** 键。例如, 为了跳转到第 100 行, 可以键入:

```
100G
```


要跳转到编辑缓冲区的开头，可以键入 **1G**。对于较新版本的 **vi** 来说，也可以使用 **gg** 代替 **1G**。

另外，也可以通过键入:(冒号)，后面跟行号，再按<Return>键跳转到指定行。下面举一些例子。其中，第一条命令跳转到第 1 行，第二条命令跳转到第 100 行，最后一条命令跳转到文件的末尾(当指定行号时，**\$**字符代表文件的最后一行)。

```
:1
:100
:$
```

下面是所有变体的小结：

```
nG  跳转到第 n 行
1G  跳转到编辑缓冲区的第一行
gg  跳转到编辑缓冲区的第一行
G   跳转到编辑缓冲区的最后一行
:n  跳转到第 n 行
:1  跳转到编辑缓冲区的第一行
:S  跳转到编辑缓冲区的最后一行
```

G 和 **1G**(或者 **gg**)命令特别有用，所以现在就应该练习使用它们并记住它们。

22.20 插入文本

正如本章前面所讨论的，为了在编辑缓冲区中插入文本，必须键入命令从命令模式切换到输入模式。当结束文本插入之后，还要按<Esc>键离开输入模式返回命令模式(记住：当在命令模式中按<Esc>键时，**vi** 会发出嘀嘀声。如果不确定处于哪一种模式中，可以按<Esc>键两次。当听到嘀嘀声时，就意味着处于命令模式中了)。

有 12 种命令可以改变到输入模式。其中一半用于输入新数据，另一半用于替换已有文本。当然，您可能会问，只是为了改变到输入模式，为什么需要如此众多的命令呢？答案是每条命令在不同的位置打开编辑缓冲区。因此，当希望插入数据时，可以选择最适合当前情形的命令进行插入。下面就是这些命令：

- i** 改变到输入模式：在当前光标位置前插入数据
- a** 改变到输入模式：在当前光标位置后插入数据
- I** 改变到输入模式：在当前行开头处插入数据
- A** 改变到输入模式：在当前行末尾处插入数据
- o** 改变到输入模式：在当前行下面插入一行
- O** 改变到输入模式：在当前行上面插入一行

为了说明这些命令的作用，假设您正在编辑一篇高级古典音乐课程的学期论文。您正

在写著名的歌词^{*}，当前行正好是：

```
For a dime you can see Kankakee or Paree
```

现在光标位于字母“K”之下，并且处于命令模式中。如果键入 **i**，将切换到输入模式。在键入时，数据将插在字母“K”之前，右边的字母自动依次右移。例如，假设您键入：

```
iAAA<Esc>
```

(按下<Esc>键后返回到命令模式。)

那么当前行看起来会像这样：

```
For a dime you can see AAANKankakee or Paree
```

现在，假设按下 **a** 键切换到输入模式。在这种情况下，数据将插入到“K”之后。因此，假设您从原始行开始，键入了下述内容：

```
aBBB<Esc>
```

那么当前行看起来会像这样：

```
For a dime you can see KBBBankakee or Paree
```

提示

要记住 **i** 命令和 **a** 命令之间的区别，可以这样想：**i**=insert(插入)，**a**=append(追加)。

通过使用 **I**(大写字母“I”)和 **A**(大写字母“A”)命令，可以分别在当前行的开头或末尾插入数据。例如，假设以原始行开始，并键入：

```
ICCC<Esc>
```

那么当前行看起来就像这样：

```
CCCFor a dime you can see Kankakee or Paree
```

如果当前行使用空格或制表符缩进了，那么 **vi** 也会聪明地考虑到这一点，在缩进位置之后开始插入数据。

现在，假设仍以原始行开始，并键入：

```
ADDD<Esc>
```

所键入的数据被追加在这一行的末尾，此时当前行看起来就像这样：

```
For a dime you can see Kankakee or PareeDDD
```

最后，为了在当前行下面插入数据，可以使用 **o**(小写字母“o”)命令。为了在当前行上面插入数据，可以使用 **O**(大写字母“O”)命令。不管是哪一种情况，**vi** 都会打开一个新行。

^{*} 这一行取自 1939 年的歌曲“Lydia the Tattooed Lady”，这首歌由 Harold Arlen 和 Yip Harburg 创作。多年以来，这首歌曲由 Groucho Marx 唱响，Groucho Marx 喜欢即兴演唱这首歌。

提示

要记住 **o** 和 **O** 命令之间的区别，可以依赖下述两点：

第一，字母“o”代表“open，打开”。

第二，将命令名称想象成一个充满氦气的气球。大一点的气球，即 **O**，浮得高，因此它在当前行的上面插入一行。而小一点的气球，即 **o**，浮得低，因此在当前行的下面插入一行。

在输入模式中时，有两件事需要记住。正如本章前面讨论的：

- 可以使用图 22-6 中所列举的键对错误进行纠正，而不必离开输入模式：使用 **<Backspace>** (或 **<Delete>**) 键删除一个字符，使用 **^W** 删除一个单词，使用 **^X** (或 **^U**) 删除整行。
- 可以通过在控制字符前面加上一个 **^V** 来插入一个控制字符。例如，为了输入一个退格，需要键入 **^V^H**。在屏幕上，看到的是 **^H**，但是它表示一个字符。

正如前面所解释的，有许多命令可以移动光标。其中，**^** (音调符号) 命令可以将光标移动到当前行的开头 (在缩进之后)；**\$** (美元符号) 命令可以将光标移动到当前行的末尾。因此，如果希望在当前行的开头插入数据，则可以键入 **^**，后跟 **i**，来取代 **I** 命令。同样，也可以使用 **sa** 取代 **A** 命令，在行的末尾插入数据。

这里描述了 **vi** 的设计之美。通过学习一些额外的命令，经常可以用一个字符的命令 (**I** 或 **A**) 来取代两个字符的命令 (**^i** 或 **sa**)。如果您是一名初学者，那么您可能会对此不以为然，但过不了多久，您就会发现为减少键入命令时的击键次数而设计的所有命令，用起来确实是很方便的。当然，您也不得不去学习这些额外的命令，这就是我们说 **vi** 用起来容易，但是学起来难的原因。

如果您习惯于使用鼠标进行编辑，那么请不要去嘲笑 **vi** 面向命令的过时设计。花些时间学习 **vi** 所有的重要命令，一旦学会之后，您就会高兴地发现在编辑数据时无需将手离开键盘去移动鼠标或者按特殊键是多么的方便。此外，您还会发现使用 **vi** 强有力的光标移动命令比用鼠标单击滚动条*要容易得多，也快得多。

提示

开始很容易使用的工具通常过一个月之后就显得过于笨拙了。

22.21 修改文本

在上一节中，讨论了可以切换到输入模式的命令，从而可以在编辑缓冲区中插入数据。在本节中，我们将讨论如何修改编辑缓冲区中已有的数据。首先，我们讨论 7 条 **vi** 命令。

* 通过观察某人使用鼠标的方式可以判断他思考的速度。思维转动得越快 (称为“ideaphoria”) 的人，越愿意使用键盘，而不是鼠标。通常，高 ideaphoria 的人不愿意将手离开键盘，因为这样会降低速度。

您见过他人阅读网页吗？高 ideaphoria 的人将按 **<PageUp>**、**<PageDown>** 或者 **<Space>** 键，低 ideaphoria 的人将使用鼠标上下移动滚动条。

除了一种情况外，其他情况都需要切换到输入模式。下面先从这个不需要切换到输入模式的命令入手。

为了用一个字符替换另一个单独的字符，可以键入 **r**，后面跟着新字符。例如，假设您正向教授写信，解释无法按时完成学期论文的原因。您现在位于命令模式中，而当前行是：

```
would mean missing The Sopranos rerun. I gm sure you
```

您注意到单词“gm”有错误。将光标移动到“g”上，然后键入：

```
ra
```

当前行将被修改为：

```
would mean missing The Sopranos rerun. I am sure you
```

因为您只修改一个字符，所以没必要进入输入模式。

假定您希望重写不止一个字符，那么首先应将光标移动到希望开始替换的位置，然后键入 **R**(大写字母“R”)。您将切换到输入模式，随后键入的每个字符都将替换当前行上的一个字符。当结束键入后，可以按<Esc>键返回到命令模式。下面举例说明。当前行如上述例子所示，将光标移动到“The”中的“T”处，然后键入：

```
RMa's funeral<Esc>
```

当前行现在变成了：

```
would mean missing Ma's funeral. I am sure you
```

当使用 **R** 命令替换文本时，**vi** 不会离开当前行。因此，如果键入的字符超过了行尾，那么 **vi** 只是在这一行的末尾追加额外的字符。

有时候，可能希望替换数据中的一个或多个字母，用来替换的数据的长度与其并不相同。这种情况下有许多命令可以使用。**s**(substitute, 替换)命令允许使用多个字符替换一个单独的字符。在我们的例子中，将光标移动到“Ma”中的 **a** 处，然后键入：

```
s
```

a 将变成一个 **\$**，而且您将位于输入模式中。这一行将变成：

```
would mean missing M$'s funeral. I am sure you
```

\$显示哪个字符将被替换。键入自己希望的字符之后，可以按<Esc>键结束键入。假设您键入了：

```
other<Esc>
```

现在当前行变为：

```
would mean missing Mother's funeral. I am sure you
```

C(大写字母“C”)命令是这种类型的改变的一种变体。它允许替换从当前光标位置到这一行末尾的所有字符。在我们的例子中,假设光标位于字母“I”处,这时键入:

C

这样将切换到输入模式,并且最后一个待替换的字符被用一个**\$**标记:

would mean missing Mother's funeral. I am sure you**\$**

当前字符是“I”。键入希望的内容,然后按<Esc>键。假设键入的内容是:

We all hoped that<Esc>

现在当前行变为:

would mean missing Mother's funeral. We all hoped that

有时候,最容易的事情就是替换一整行。有两条命令可以完成这一任务,它们分别是**S**和**cc**。只需将光标移动到希望替换的那一行,然后键入这两条命令中的一条,这样就会进入输入模式。当按<Esc>键时,所键入的内容就替换掉这一整行。

为什么两条作用相同的命令却拥有完全不同的名称呢?许多 vi 命令的名称都遵循一种命名模式。有的名称只有一个小写字母,有的是两个小写字母,有的是一个大写字母。根据这种模式,**S**和**cc**都应该是替换一整行的命令。因此,可以选择使用其中任意一个(如果现在还不能明白这种模式,不用担心。等学会了更多的命令之后就会明白)。

最后一个替换数据的 vi 命令相当有用。该命令就是**c**后面跟一个移动光标的命令。再一次说明,这样也会进入输入模式。但是,这一次键入的内容将替换从光标当前位置到由移动命令所指定的位置之间的所有字符。该命令很容易混淆,因此下面举几个例子说明一下。假设当前行是:

would mean missing Mother's funeral. We all hoped that

光标位于字母“M”处。您希望将整个单词“Mother”替换为“my dog”。因此键入:

cw

这将切换到输入模式,并用一个**\$**标记待替换的最后一个字符。这时会看到:

would mean missing Mothe**\$**'s funeral. We all hoped that

现在键入:

my dog<Esc>

当前行变为:

would mean missing my dog's funeral. We all hoped that

因此,**cw**组合可以改变一个单词。**c**可以与任意一个单字符的光标移动命令组合使用。如果愿意,还可以使用重复次数。例如,命令**c5w**替换5个单词。命令**c4b**从当前位置向

后替换 4 个单词。命令 **c**(从当前位置开始向后替换到句子的开头。命令 **C**从当前位置开始替换到段落的末尾。为了替换 6 个段落，可以将光标移动到第一个段落的开头，键入 **c6**)。

vi 替换命令总结如下：

r 精确替换 1 个字符(不进入输入模式)

R 以覆盖方式替换

s 以插入方式替换 1 个字符

C 以插入方式从当前光标处替换到这一行的末尾

cc 以插入方式替换当前的整行

S 以插入方式替换当前的整行

cmove 以插入方式从当前光标处替换到 *move* 所给出的位置处

22.22 替换文本

正如前面所述,当使用 **vi** 时,既可以使用 **vi**(面向屏幕)命令,也可以使用较古老的 **ex**(面向行)命令。到目前为止,我们讨论的大多数命令都是 **vi** 命令。在本节中,我们将讨论 **ex** 命令。

所有的 **ex** 命令都以一个:(冒号)字符开头。每当在命令的开头键入一个冒号时,**vi** 将在命令行(屏幕的底部)上立即显示它。在键入命令的其他部分时,命令将回显在这一行上(这就是称这一行为命令行的原因)。可以看出, **ex** 命令要比 **vi** 命令长,而且更复杂。基于这一原因,在键入命令的过程中,**vi** 将回显命令,从而使您可以看到自己正在做什么。**vi** 命令中仅有的复杂长命令就是搜索命令(/和?),也是由于这个原因,这两个命令也在命令行上显示。

为了替换一个特定的模式,可以使用 **ex** 命令 **:s**(substitute, 替换)。其语法为:

```
:s/pattern/replace/
```

其中 *pattern* 是希望替换的模式, *replace* 是替换文本。例如,为了将当前行上的“UNIX”替换为“Linux”,可以使用:

```
:s/UNIX/Linux/
```

以这种方式使用 **:s** 只替换当前行上该模式的第一个匹配项。为了替换所有的匹配项,需要在该命令的末尾键入字母 **g**(global, 全局)。例如,为了将当前行上所有的“UNIX”都替换为“Linux”,可以使用:

```
:s/UNIX/Linux/g
```

如果希望 **vi** 在进行改变之前先经过您的同意,则需要在该命令的末尾添加字母 **c**(confirm, 确认):

```
:s/UNIX/Linux/c
```


当然，也可以将 **g** 和 **c** 组合在一起使用：

```
:s/UNIX/Linux/cg
```

当使用 **c** 修饰符时，**vi** 将显示包含该模式的行。**vi** 指出模式的位置，然后等待确认。如果希望进行替换，可以键入 **y**(代表 **yes**)，然后再按 **<Return>** 键。否则，键入 **n<Return>**(代表 **no**)，或者只是按 **<Return>** 键本身(如果不指定 “**y**” 或 “**n**”，那么 **vi** 将慎重地假定您不希望进行修改)。

为了删除模式，只需将该模式替换为空即可。例如，为了移除当前行上所有的 “UNIX”，可以使用：

```
:s/UNIX//g
```

为了方便起见，如果不在命令的末尾使用一个 **c** 或者 **g**，则可以省略最后的 **/** 字符。例如，下述两条命令是等价的：

```
:s/UNIX/Linux/  
:s/UNIX/Linux
```

:s 命令有两个重要的变体。首先，可以在冒号后面指定一个特定的行号。这将告诉 **vi** 在特定行上执行替换。例如，为了将第 57 行上第一次出现的 “UNIX” 改变为 “Linux”，可以使用：

```
:57s/UNIX/Linux/
```

(提醒：使用 **:set number** 可以显示行号；使用 **:set nonumber** 可以隐藏行号。)

除了单独的行号之外，还可以使用两个用逗号分隔的行号表示一个范围。例如，为了在第 57 行至第 60 行之间进行相同的替换，可以使用：

```
:57,60s/UNIX/Linux/
```

在这个例子中，**vi** 将替换该范围之内每行上指定模式的第一个匹配项。

大多数时候，您不会使用具体的行号。但是，有 3 种特殊的符号使这种形式的命令特别有用。**.**(点号)代表当前行，**\$**(美元符号)代表编辑缓冲区的最后一行。因此，下述命令将当前行至编辑缓冲区末尾的所有 “UNIX” 都替换为 “Linux”：

```
:.,$s/UNIX/Linux/g
```

为了从编辑缓冲区的开头到当前行进行相同的改变，可以使用：

```
:1,.$s/UNIX/Linux/g
```

第三个特殊符号是 **%**(百分比符号)，它代表编辑缓冲区中的所有行。因此，为了将编辑缓冲区中的每一个 “UNIX” 都改变为 “Linux”，可以使用：

```
:%s/UNIX/Linux/g
```

这与从第 1 行至第 **\$** 行(编辑缓冲区的末尾)进行替换的效果相同：

```
:1,$/UNIX/Linux/g
```

使用%要比键入 1,\$方便许多,因此一定要记住这个方便的缩写,今后您会大量地使用它。

有时候,可能希望 vi 确认每个替换。这样可以控制替换模式的哪些实例。正如前面所讨论的,只需使用 c(confirm, 确认)修饰符即可,例如:

```
:%s/UNIX/Linux/cg
```

当使用这样的命令时,可以通过按^C(intr 键)在半途停止。这将终止整条命令,而不仅仅是当前替换。

出于参考目的,下面汇总了:s 命令的使用方法:

:s/pattern/replace/	替换当前行
:lines/pattern/replace/	替换指定行
:line,lines/pattern/replace/	替换指定范围的行
:%s/pattern/replace/	替换所有行

在命令的末尾,可以使用 c 告诉 vi 在替换之前要求确认,而使用 g(global, 全局)则表示替换每行上的所有匹配项。为了指定行号,可以使用实际数字,也可以使用.(点号)表示当前行,或者使用\$(美元符号)表示编辑缓冲区的最后一行。数字 1 表示编辑缓冲区的第一行。

22.23 删除文本

从编辑缓冲区中删除数据的方法有若干种,您既可以使用 vi 命令,也可以使用 ex 命令。vi 命令如下所示:

x	删除当前光标处的字符
X	删除光标左边的字符
D	删除从当前光标到本行末尾的字符
dmove	删除从当前光标到 move 所给位置的字符
dd	删除当前行

另外,还有两个 ex 命令:

:lined	删除指定行
:line,lined	删除指定范围的行

无论使用哪一条命令,都可以使用撤销命令 u 和 U(下一节讨论)撤销删除。记住:有一天该命令会帮您的大忙。

最简单的删除命令是 x(小写字母“x”)。它删除当前光标位置处的字符。例如,假设

您给父母写信，告诉他们您在学校的生活情况。编辑缓冲区中的当前行包含：

```
I love heiQnous paWrties and avoid the library as a rule
```

您发现第 3 个单词中有一个错误。您可以将光标移至“Q”处并键入：

x

现在当前行变为：

```
I love heinous paWrties and avoid the library as a rule
```

X(大写字母“X”)也删除一个字符，但不同的是它删除光标左边的字符。例如，在上述行中，您注意到在第 4 个单词中还有另一个错误。您可以将光标移动到“r”处并键入：

x

现在当前行变为：

```
I love heinous parties and avoid the library as a rule
```

D(大写字母“D”)命令删除从当前光标位置到这一行末尾的所有字符。例如，假设您将光标移动到单词“library”之后的空格处，并键入：

D

当前行将变为：

```
I love heinous parties and avoid the library
```

下一条删除命令是 **d**(小写字母“d”)，后面跟一条光标移动命令。该命令将删除从当前光标位置到光标移动命令所指示位置的所有字符。这与前面讨论的 **c**(change, 改变)命令相似。下面举一些例子：

```
dw    删除 1 个单词
d10w  删除 10 个单词
d10W  删除 10 个单词(忽略标点符号)
db    向后删除 1 个单词
d2)   删除 2 个句子
d5}   删除 5 个段落
```

提示

d 命令有两种特别有用的使用方法。第一，为了删除当前行到编辑缓冲区末尾的所有行，可以使用 **dG**。

第二，为了删除从当前行到编辑缓冲区开头的所有行，可以使用 **dgg** 或者 **d1G**(正如前面所述，**gg** 命令不适用于老版本的 vi)。

下面继续我们的例子，当前行仍然是：

I love heinous parties and avoid the library

假设将光标移动到单词“heinous”的开头，并通过键入下述命令删除 4 个单词：

d4w

现在当前行变为：

I love the library

最后一条 **vi** 删除命令是 **dd**。该命令删除当前行。如果希望删除不止一行，可以在该命令前面加上一个重复次数。例如，为了删除一行，可以使用：

dd

为了删除 10 行，可以使用：

10dd

有时候，使用行号进行删除可能更方便些。此时，可以使用 **ex** 命令 **:d**。为了使用 **:d** 命令，需要指定一个行号或者一个范围(两个行号，中间用一个逗号隔开)。例如，为了删除第 50 行，可以使用：

:50d

为了删除第 50 行至第 60 行，可以使用：

:50,60d

(提醒：为了显示行号，可以使用 **:set number**；为了关闭行号的显示，可以使用 **:set nonumber**。)

和其他 **ex** 命令一样，符号 **.**(点号)代表当前行，**\$**(美元符号)代表编辑缓冲区中的最后一行。因此，为了将编辑缓冲区的第 1 行到当前行之间的所有行删除，可以使用：

:1,.d

这与 **dgg** 和 **d1G** 命令拥有相同的结果。为了从当前行删除到编辑缓冲区的末尾，可以使用：

:\$d

这与 **dG** 命令拥有相同的结果。为了删除整个编辑缓冲区，可以使用下述命令中的一个：

:1,\$d

:%d

记住，**%**代表编辑缓冲区中的所有行。

22.24 撤销或重复改变

一旦开始进行替换和删除，能够撤销改变就变得十分重要。例如，假设您希望将单词“advertisement”的所有匹配项都改变为“ad”。您决定输入：

```
:%s/advertisement/ad/g
```

但是，您不小心键入错误，忘记了键入第二个“d”：

```
:%s/advertisement/a/g
```

这样就将所有的“advertisement”都替换为字母“a”。但是，您不能通过将所有的“a”改变为“ad”来解决这个问题，因为字母“a”到处都是。您可以使用`:q!`命令，不保存工作就退出 vi(如果处理的是一个已有文件的话)，但是这样就会丢失整个编辑期间的所有改变，前面的工作都白白浪费。那么，还有没有其他的方法？

下面是一个相似的案例。您希望删除 10 行，但是您键入的不是 `10dd`，而是 `100dd`，结果您删除了 100 行。有没有方法将这 100 行恢复呢？

两个问题的答案都是肯定的。vi 中有两条命令可以用来撤销改变，另外还有一条重复上一次改变的命令：

- u** 撤销上一命令对编辑缓冲区的修改
- U** 恢复当前行
- .** 重复上一命令对编辑缓冲区的修改

u(小写字母“u”)命令撤销上一命令对编辑缓冲区所做的改变：插入、替换、改变或删除。在两个例子中，只需键入 **u**，所有的替换/删除都会被废除。如果在按了 **u** 键之后，您又决定保留原来的改变，则只需再次按下 **u** 键。**u** 命令可以撤销自己(要是生活能这样，那该有多好！)*

U(大写字母“U”)命令将撤销自移动到当前行的那一刻起，对当前行所进行的所有改变。例如，假设您将光标移动到某一特定行，并进行了大量的改变，但是没有离开这一行。然而，您将这一行搞乱了，因此您决定将这一行恢复到刚刚移动到这一行时的模式。这时只需键入 **U**，这一行将恢复其原始内容。如果在按了 **U** 键之后，又不喜欢这种结果了，可以使用 **u**(小写字母“u”)撤销该命令。

U 命令可以撤销大量的改变，一次就将当前行的所有改变都恢复。但是，仅当还没有离开这一行时 **U** 命令才有效。一旦将光标移到新行上去，**U** 命令就只适用于这个新行，此时将无法找到简单的方法来恢复旧行上所做的修改。

除了 **u** 和 **U** 命令之外，还有另外一条重要的命令，该命令涉及到对编辑缓冲区的上一次改变。这条命令就是`.`(点号)命令。使用它可以重复上一条命令对编辑缓冲区所做的修改。该命令非常有用，因此下面将举例说明。

* 对于 vi 来说，生活才如此简单。对于 Vim 来说，连续按 **u** 键多次将撤销前面多条命令，每次一条。当然，在许多情况下，这可能正好就是所需的结果。

假设您希望在编辑缓冲区的几个不同位置插入名称“Mxyzptlk”。这是一个比较难拼写的名称，而且每次插入都重新键入也很烦人，但是有一种巧妙的方法可以实现这一目标。将光标移动到希望进行第一次插入的位置上，然后键入：

```
iMxyzptlk<Esc>
```

您已经将这一名称插入到编辑缓冲区。现在，将光标移动到下一个希望插入的位置上去，并键入：

```
.
```

这将执行完全相同的插入。您可以根据需要多次使用.命令，但是一定要小心：一旦进行了另一个改变，即便只是一个字符的删除，.命令的结果也将会随之改变。

22.25 恢复删除

每当删除一行或多行文本时，vi 都将删除内容保存在一个特殊的存储区中，这个存储区称为编号缓冲区(numbered buffer)。vi 中共有 9 个这样的缓冲区，编号从 1 至 9。在任何时候，都可以将一个编号缓冲区中的内容插入到编辑缓冲区中。这样做时，需要键入一个“(双引号)，后面跟着缓冲区的编号，再后跟一个 **p** 或者 **P**(put, 放入)命令(提醒：当处理行时，**p** 命令在当前行的下面插入，**P** 命令在当前行的上面插入)。

例如，为了将编号缓冲区#1 的内容插入到当前行的下面，可以使用：

```
"1p
```

为了将编号缓冲区#2 的内容插入到当前行的上面，可以使用：

```
"2P
```

通过这种方式，可以恢复并插入之前 9 次删除的内容。例如，假设您已经进行了若干次删除，而您希望恢复其中一次删除。但是，您记不清是哪一次了。

首先键入"**1p**，如果所恢复的文本不是所需的，则可以键入 **u** 撤销插入，并尝试"**2p**。如果这一次又未得到所需的文本，可以继续键入 **u** 撤销这次插入，再尝试"**3p**。一直这样往下试，直到找到自己希望的内容。命令序列看上去如下所示：

```
"1pu"2pu"3pu . . .
```

该命令本身相当的酷，但是，vi 比这个更强大。在撤销了第一次插入之后，如果使用.(点号)命令重复插入，那么 vi 将自动地将缓冲区编号加 1。这意味着，不用再使用上一个命令序列，可以使用：

```
"1pu.u.u . . .
```

为了测试这个命令序列，先使用 vi 创建一个包含如下 5 行内容的文件：

```
111
```

```
222
```


333

444

555

键入 **1G** 或 **gg** 跳转到第一行。然后，连续键入 **dd** 命令 5 次删除所有行。现在就可以试一试上述恢复命令序列，看看会发生什么情况。

提醒：编号缓冲区只存储删除的行，而不是行的一部分或者单个字符。例如，如果使用 **10dd** 删除 10 行，那么该删除将存储在一个编号缓冲区中。但是，如果使用 **5x** 删除 5 个字符，那么该删除不会保存在编号缓冲区中。

22.26 移动文本

vi 编辑器拥有一项特殊的功能，即将文本从一个位置移动或者复制到另一个位置上。在本节中，我们讨论文本的移动。在下一节中，我们将讨论文本的复制。

vi 总是在一个称为无名缓冲区(unnamed buffer)的存储区中为上一次删除保存一份副本。在任何时候，都可以使用 **p** 和 **P**(put, 放入)命令将无名缓冲区中的内容复制到编辑缓冲区中(这块存储区被称为无名缓冲区的原因在于还有一些其他有名称的类似存储区)。

p(小写字母“p”)命令将无名缓冲区中的内容插入到光标的当前位置之后。例如，假设当前行包含：

```
This good is a sentence.
```

将光标移动到“g”上，并删除一个单词。

```
dw
```

在这样做时，删除的单词将被复制到无名缓冲区中。现在当前行变为：

```
This is a sentence.
```

下面将光标移动到“a”之后的空格处，并键入：

```
P
```

无名缓冲区的内容将插入到光标的右边。现在当前行变为：

```
This is a good sentence.
```

下面举一个使用 **P**(大写字母“P”)命令的例子。假设当前行包含：

```
This is right now.
```

将光标移动到单词“right”前面的空格处，并键入：

```
de
```

这将删除这个单词，结果如下所示：

```
This is now.
```

现在将光标移动到行尾的点号处，并键入：

P

被删除的单词将插入到光标的左边。现在当前行变为：

This is now right.

无名缓冲区每次只能存放一次删除的内容，理解这一点非常重要。例如，假设您刚删除了 1000 行文本。无名缓冲区中存储了该文本的一份副本。如果希望，则可以将该文本插入到编辑缓冲区的任何位置上。现在，您又删除了一个字符。这时候，无名缓冲区中的 1000 行文本将被移除。如果现在再使用 **p** 命令，那么得到的是最后一次删除的内容，也就是一个字符。

下面考虑命令组合 **xp**。**x** 命令删除当前光标位置处的字符。**p** 命令将删除的内容插入到光标的右边。该组合命令的最终结果就是调换两个字符的位置。例如，假设当前行为：

I ma never mixed up.

将光标移动到第一个“m”处并键入：

xp

现在当前行变为：

I am never mixed up.

另一个重要的组合命令是 **deep**，该组合命令可以用来调换两个单词。下面举例说明，假设当前行包含：

I am mixed never up.

将光标移动到单词“mixed”前面的空格处(一定要注意将光标移动到这个单词前面的空格处，而不是这个单词的第一个字母上)。现在键入：

deep

de 命令删除空格及后面的单词，这时当前行变为：

I am never up.

第二个 **e** 命令移动到下一个单词的尾部，然后 **p** 命令将删除的内容插入到光标之后。最终结果是：

I am never mixed up.

通过这种方式，可以轻松地调换两个单词。考虑一下，并记住这条组合命令，以便在需要时，可以快速键入这条组合命令。

每当删除整行时，**p** 和 **P** 也将插入整行。不同的是，**p** 命令将删除的行插入到当前行的下面，而 **P** 命令将删除的行插入到当前行的上面。例如，假设您希望从一个位置向另一个位置移动 10 行文本。首先，将光标移动到文本的第一行。然后使用一个带重复次数的

dd 命令删除 10 行:

```
10dd
```

这些行将从编辑缓冲区中被删除, 并复制到无名缓冲区中。下面将光标移动至在其下面进行插入的那一行上, 并键入:

```
p
```

现在, 考虑如果键入 **ddp** 会发生什么情况。**dd** 命令删除当前行。下一行成为新的当前行。**p** 命令将删除内容插入到新当前行的下面。最终结果是调换了两行(试试看)。

概括起来:

```
p      复制上一次删除的内容, 插入到光标后面/下面
P      复制上一次删除的内容, 插入到光标前面/上面
xp     调换两个字符
deep   调换两个单词(光标开始处于第一个单词的左边)
ddp    调换两行
```

22.27 复制文本

从一个位置向另一个位置复制文本包括 3 步。第一步, 使用 **y**、**yy** 或 **Y** 命令将文本由编辑缓冲区复制到无名缓冲区中, 但不删除原始文本。第二步, 将光标移动到希望插入文本的位置。第三步, 使用 **p** 或 **P** 命令执行插入。

当在不删除文本的情况下将文本复制到无名缓冲区时, 我们称接出(yank)了文本(因此将命令命名为 **y**、**yy** 和 **Y**)。**y** 和 **yy** 命令的工作方式与 **d** 和 **dd** 命令的工作方式相同, 只是 **y** 和 **yy** 命令接出文本, 而 **d** 和 **dd** 命令删除文本。下面示范几个例子:

```
yw      接出 1 个单词
y10w    接出 10 个单词
y10W    接出 10 个单词(忽略标点符号)
yb      向后接出 1 个单词
y2)     接出 2 个句子
y5)     接出 5 个段落
yy      接出 1 行
10yy    接出 10 行
```

假设您希望从一个位置向另一个位置复制 5 个段落的文本。首先, 将光标移动到第一段的开头。接下来, 将 5 个段落的文本接出到无名缓冲区中, 但不删除文本:

```
y5)
```

下面将光标移动到希望在其下进行插入的那一行文本上, 并插入文本:

P

为了方便起见，可以使用 **Y** 替代 **yy**。因此，下面的命令都将 10 行文本接出到无名缓冲区中：

```
10yy
10Y
```

注意有一些事情特别有趣。**y** 命令与 **d** 命令类似，它们都将从当前字符到光标移动命令指定的字符之间的文本复制到无名缓冲区中(唯一的区别就是 **d** 删除文本，而 **y** 接出文本)。同样，**yy** 与 **dd** 类似，它们分别删除/接出整行。

但是，**Y** 命令并不与 **D** 命令类似。**Y** 命令接出整行，而 **D** 命令删除从当前字符到这一行末尾之间的文本。如果希望接出从当前字符到这一行末尾之间的文本，则必须使用 **y\$**。为了接出从当前字符到这一行开头之间的文本，可以使用 **y0**。

提示

每当删除或接出文本时，文本一直保留在无名缓冲区中，直至您输入另一条删除或接出命令。因此，可以使用 **p** 或 **P** 命令，在编辑缓冲区的不同位置上一遍又一遍地插入相同的文本。

22.28 改变字母的大小写

vi 编辑器拥有一个特殊的命令，可以将字母由小写字母变成大写字母，或者从大写字母变成小写字母。该命令就是 **~**(波浪号)。只需将光标移动到希望改变的字母上，并键入：

~

~命令将致使 **vi** 改变当前字符的大小写，然后将光标向前移一个位置。例如，假设当前行包含：

```
"By Jove," he said, "that's a CAPITAL idea."
```

现在光标位于“C”上。按下**~**键之后，当前行变为：

```
"By Jove," he said, "that's a cAPITAL idea."
```

光标现在位于“A”上。因为**~**将光标向右移一个位置，所以可以重复地键入**~**改变一串字母的大小写。在我们的例子中，通过再键入 6 个**~**字符就可以将这个单词的剩余部分变成小写字母：

```
~~~~~
```

现在当前行变为：

```
"By Jove," he said, "that's a capital idea."
```

当光标位于不是字母的字符(例如标点符号)处时,如果键入~,vi将使光标向前移动一个位置,但不进行改变。因此,在大范围内连续使用“波浪号”是安全的,因为vi将简单地跳过非字母表字符。为了更加容易,可以在该命令前面使用一个重复次数。例如,为了改变一个7个字母的单词的大小写,可以将光标移动到这个单词的开头,并键入:

```
7~
```

这样整个单词的大小写都将改变,最终光标将位于这个单词末尾的后面一个位置上。

注意:对于一些版本的vi来说,~命令不会越过一行的末尾,即使是使用一个大的重复次数,例如100~。其他版本的vi能够处理任意多的字符,甚至跨行进行处理。当有空闲时间时,最好体验一下自己的vi版本,查看~命令会不会越过一行的末尾。

22.29 设置选项

和大多数复杂的Unix程序一样,vi支持许多选项,从而允许用户控制vi操作的各个方面。当启动vi时,每个选项都被赋予一个默认值。如果希望改变vi行为的一个特定方面,则可以使用:set命令设置合适选项的值。该命令的语法有两种形式,因为选项本身就有两种不同类型:

```
:set [no]option...
:set option[=value]...
```

其中option是选项的名称,value是选项的值。

在大多数情况下,默认值就能够满足需求(所以称之为默认值)。但是,有时候,也可能希望做些改变。我们在本章中已经进行了3次改变。首先,我们使用showmode选项告诉vi当进入输入模式时显示一个提醒。使用的命令是:

```
:set showmode
```

我们使用过的第二个选项是number,该选项用来显示行号:

```
:set number
```

最后一个使用过的选项是nonumber,该选项用来关闭行号的显示:

```
:set nonumber
```

vi的选项有两种类型。第一种类型的选项是开关(switch),这种选项的值或者为关闭(off),或者为打开(on)。前面提到了几个选项都是开关型选项。为了打开一个开关,需要使用该开关的名称;为了关闭一个开关,需要在该开关的前面键入“no”。例如:

```
:set showmode
:set noshowmode
:set number
:set nonumber
```

第二种类型的选项是变量(variable)，这种类型的选项包含一个值。例如，**tabstop** 选项用来设置制表符的间距。默认情况下，**tabstop** 设置为 8，这意味着制表符的间距为 8 个位置(Unix 通常采用这种设置，参见第 18 章)。如果希望制表符的间距是 4 个位置，可以将 **tabstop** 变量设置为 4：

```
:set tabstop=4
```

为了方便起见，可以在同一个命令中设置多个选项，例如：

```
:set showmode nonumber tabstop=4
```

实际可用的选项取决于所使用 **vi** 的版本。可以预见，所使用的版本越新，拥有的选项就越多。一般情况下，标准 **vi** 拥有大约 40 个选项，其中有 16 个选项非常重要。**Vim** 拥有的选项超过了 340 个，但几乎所有的选项都永远不需要。出于参考目的，图 22-9 列举了重要的 **vi** 选项。

开关	缩写	默认值	含义
autoindent	ai	off	和 shift width 相关，缩进以匹配上一行/下一行
autowrite	aw	off	如果文本已经修改，则在切换文件前保存
errorbells	eb	off	当显示错误消息时发出嘀嘀声
exrc	ex	off	在当前目录中查找初始化文件
ignorecase	ic	off	在搜索过程中忽略大小写
list	—	off	将制表符显示为 [^] I，将行的结束显示为\$
number	nu	off	显示行号
readonly	ro	off	不允许修改编辑缓冲区的内容
showmatch	sm	off	输入模式：显示匹配的()、{}或[]
showmode	smd	off	当进入输入模式时显示一个提醒
wrapscan	ws	off	在搜索过程中，环绕到文本的开头/末尾继续搜索
writeany	wa	off	允许不需要重载！就可以写入任何文件

变量	缩写	默认值	含义
lines	—	24	文本的行数(窗口/屏幕大小-1)
shiftwidth	sw	8	autoindent 使用的空格数量
tabstop	ts	8	制表符间距
wrapmargin	wm	0	设置自动换行时的页边距(0=off)

图 22-9 vi 选项：开关和变量

有两种不同类型的选项可以用来控制 **vi** 的各个不同方面：开关(关闭或打开)和变量(存储一个值)。这两个列表列举了最有用的开关和变量，以及它们的缩写、默认值和含义。

从图中可以看出，几乎所有的选项都有缩写。为了方便起见，可以使用缩写取代全称。

例如，下述两条命令是等价的：

```
:set showmode nonumber tabstop=4
:set smd nonu ts=4
```

提示

为了在每次启动 vi 时自动地设置选项，可以将合适的:set 命令放在初始化文件中(详见本章后面的讨论)。

22.30 显示选项

为了显示一个或多个选项的值，可以使用:set 命令的一种变体。其语法为：

```
:set [option[?]]... | all]
```

其中 *option* 是选项的名称。

为了显示所有选项的值，可以使用：

```
:set all
```

使用这条命令是查看 vi 所支持的全部选项的最佳方法。为了显示一个选项的值，可以键入该选项的名称，后面跟一个?(问号)。例如：

```
:set number?
:set showmode?
:set wrapmargin?
```

为了方便起见，可以在一条命令中显示多个选项的值：

```
:set number? showmode? wrapmargin?
```

最后，为了显示那些改变了默认值的选项，可以使用:set 命令本身：

```
:set
```

当使用最后一条命令时，可以查看修改了哪些选项，其中一些修改您可能并不知晓。这是因为每个系统中，vi 在启动时都要读取许多初始化文件。一些文件可以由我们控制，用来设置工作环境，以适合我们的需求(我们将在本章后面讨论)。其他初始化文件或者在 vi 安装时自动创建，或者由系统管理员建立。一般情况下，这些文件包含有修改特定选项的值的命令，这些选项也会在拥有非默认值选项的列表中列举。

22.31 在键入过程中自动换行

当键入文档时，需要将文本分成行。一种方式就是在每行的末尾按<Return>键。正如第 7 章中解释的，按<Return>键生成一个新行字符，标记一行的结束。这对于处理少量的文本比较适用，但是如果处理时需要大量的键入，让 **vi** 自动换行会更加方便。为此，需要设置 **wrapmargin(wm)** 选项。其语法为：

```
:set wrapmargin=n
```

其中 *n* 是从右边缘算起希望开始换行的位置。为了方便起见，可以使用缩写 **wm** 取代全称。

wrapmargin 选项只影响输入模式。将 **wrapmargin** 选项设置成一个大于 0 的数以后，在键入过程中，当行离右边缘达到指定字符数时，就会使 **vi** 自动换行。例如，为了告诉 **vi** 在离右边缘 6 个字符时自动换行，可以使用下述两条命令中的一条：

```
:set wrapmargin=6
:set wm=6
```

如果希望文本行尽可能的长，则可以将该选项的值设置为 1：

```
:set wm=1
```

要关闭自动换行，可以将 **wm** 设置为 0：

```
:set wm=0
```

如果希望文本缩进，则可以打开 **autoindent(ai)** 选项：

```
:set autoindent
```

这将告诉 **vi** 匹配正在键入的行相对于上一行或下一行的缩进。

自动换行只影响正在键入的文本。为了重新格式化现有的文本，可以使用 **r** 和 **J** 命令(参见下一节)，或者第 18 章中的 **fmt** 命令(本章稍后讨论)。

提示

如果 **wrapmargin** 选项的值被设置为小于 6，那么每行末尾只有极小的空间，这会使修订非常困难。依我个人的经验来看，**wrapmargin** 选项的最佳值应位于 6 和 10 之间，从而为小量的修改留下足够的空间。

22.32 分隔与连接行

有时候，需要将长行分隔成两行，或者将两个短行连接在一起形成一个长行。例如：

```
This line is much too long and must be broken into two.
```

假设您希望在单词“and”处将这一行分隔。最简单的方法就是将光标移动到“and”之后的空格处，然后键入：

```
r<Return>
```

使用 **r** 命令可以将一个字符替换为另一个字符。在这个例子中，**r** 命令使用新行字符取代空格，从而将行分隔。

如果有一些行非常短，则可以将光标移动到第一行，并键入 **J**(大写字母“J”)将行合并。这将把当前行和下一行合并成一个长行。当 **vi** 连接行时，它会在合适的位置自动地插入空格，即在单词之间插入一个空格，在句子末尾插入两个空格(难道这样不漂亮吗?)

为了一次将超过两行的行连接在一起，可以在 **J** 命令之前放置一个重复次数。下面举例说明。假设编辑缓冲区中包含下述行：

```
This sentence
is short.
This sentence is also short.
```

将光标移至第一行，然后键入：

```
3J
```

结果是：

```
This sentence is short. This sentence is also short.
```

提示

r 命令和 **J** 命令在进行小调整方面非常有用。但是，当需要重新格式化超过 5~6 行的文本时，通常最好使用 **fmt** 命令，该命令在本章后面解释。

22.33 复制与移动行

有时候，通过指定行号来复制或移动行是比较方便的。对于这些操作，可以使用 **ex** 命令：**co**(copy, 复制)和**m**(move, 移动)。这两条命令使用相同的格式。唯一的区别就是**m**删除原始行，而**co**不删除原始行。

为了使用这些命令，需要在命令名称前面指定一个单独的行号或者一个行号范围。这些就是要复制或移动的行。在命令名称之后，可以指定目标行号。新行将插入到目标行的下面。命令的语法为：

```
x[,y]:coz
x[,y]:mz
```

其中 *x*、*y* 和 *z* 都是行号。

源行(*x*，或 *x* 至 *y*)被复制或移动，并插入到目标行(*z*)的下面。下面举几个例子说明：

```
:5co10    复制第 5 行，插入到第 10 行下面
:4,8co20  复制第 4 行至第 8 行，插入到第 20 行下面
:5m10     移动第 5 行，插入到第 10 行下面
:4,8m20   移动第 4 行至第 8 行，插入到第 20 行下面
```

(提醒: 为了显示行号, 可以使用 **:set number** 命令; 为了关闭行号的显示, 可以使用 **:set nonumber** 命令。)

和其他 **ex** 命令一样, 可以使用 **.**(点号)代表当前行, 使用 **\$**(美元符号)代表编辑缓冲区中的最后一行。例如, 下述命令将第 1 行至当前行之间的内容移动到编辑缓冲区的末尾:

```
:1,.m$
```

另外也可以使用行 0(零)代表编辑缓冲区的开头。例如, 下述命令将当前行至最后一行的内容移动到编辑缓冲区的开头:

```
:.,$m0
```

上面两条命令非常有趣。它们都交换编辑缓冲区的上面和下面两部分。但是, 二者之间存在一个微小的区别。对于第一条命令来说, 当前行成为编辑缓冲区的最后一行。对于第二条命令来说, 当前行是编辑缓冲区的第一行(好好地想一想)。

22.34 输入 shell 命令

vi 中有几种调用常规 shell 命令的方法, 而且在调用时不用停止 **vi**。第一种, 可以通过键入 **!:**, 后面跟希望运行的命令来输入命令。这将告诉 **vi** 将命令发送给 shell 执行。当命令结束后, 控制将返回给 **vi**。例如, 为了显示时间和日期, 可以输入:

```
!:date
```

在命令执行结束后, 可以看到一个消息。该消息根据 **vi** 版本的不同而有所不同。下面示范几种常见的消息:

```
Press ENTER or type command to continue
[Hit return to continue]
Press any key to continue
```

此时, 只需简单地按下 **<Return>** 键就可以返回到 **vi** 中。为了重复执行最近一条 shell 命令, 不管上次输入命令以来过了多长时间, 都可以使用下述命令完成该任务:

```
:!!
```

例如, 如果上一次输入的 shell 命令是 **date**, 那么使用 **:!!** 就可以再次显示时间和日期。

为了在编辑缓冲区中直接插入 shell 命令的输出, 可以使用 **:r!** 命令。我们将在下一节中讨论这一条命令, 并且示范一些例子。

有时候，可能希望输入不止一条 shell 命令。这时可以启动一个新的 shell。有两种方法，第一种就是使用 **:sh** 命令：

```
:sh
```

这将暂停 **vi**，并启动一个新的默认 shell 副本。然后，您就可以根据需要输入任意数量的命令。当使用完这个 shell 之后，可以按 **^D** 键或输入 **exit** 命令停止这个 shell。这样将又返回到 **vi** 中。

另外，还可以通过运行一条实际命令来启动一个新 shell。例如，为了启动一个新的 Bash shell，可以运行 **bash** 程序：

```
:!bash
```

为了启动一个新的 Tcsh shell，可以使用：

```
:!tcsh
```

(各种不同的 shell 参见第 11 章中的讨论。)

和 **:sh** 命令一样，当结束 shell 时系统将返回到 **vi** 中。当基于某些原因希望使用非默认的 shell 时，这种技术就比较方便。例如，假设您通常使用 **Bash**，但是这一次，您希望运行 **Tcsh** 来测试某些事情。您所需做的就是使用命令 **:!tcsh**。

概括起来：

:!command	暂停 vi ，执行 shell 命令
:!!	暂停 vi ，执行前一条 shell 命令
:sh	暂停 vi ，启动一个新 shell(默认 shell)
:!bash	暂停 vi ，启动一个新 Bash shell
:!tcsh	暂停 vi ，启动一个新 Tcsh shell

22.35 将文件中的数据插入到编辑缓冲区中

为了从现有的文件向编辑缓冲区中读入数据，可以使用 **:r** 命令。该命令的语法为：

```
: [line]r file
```

其中 *line* 是行号，*file* 是文件的名称。

:r 命令读取文件的内容，并将该内容插入到编辑缓冲区中的指定行之后。例如，下述命令将文件 **info** 的内容插入到第 10 行之后：

```
:10r info
```

使用行 0(零)可以表示编辑缓冲区的开头。例如，为了将 **info** 的内容插入到编辑缓冲区的开头，可以使用：

```
:Or info
```

使用\$可以表示编辑缓冲区的末尾。例如，为了将 **info** 的内容插入到编辑缓冲区的末尾，可以使用：

```
:$r info
```

如果省略了行号，那么 **vi** 将把新数据插入到当前行之后。这可能是最有用的:**r** 命令形式。只需将光标移动到希望插入新数据的位置上，然后输入:**r** 命令即可。例如，假设您希望将文件 **info** 的内容插入到当前段落的末尾。使用}(右花括号)命令可以跳转到段落的末尾，然后就可以插入数据：

```
}  
:r info
```

22.36 将 shell 命令的输出插入到编辑缓冲区中

:**r** 命令有一种特别有用的变体。如果在:**r** 命令之后不是键入文件的名称，而是键入一个!(感叹号)，后面跟一个程序的名称，那么 **vi** 将执行该程序，并将程序的输出插入到编辑缓冲区中。下面举例说明。**ls** 程序显示工作目录中的文件列表(参见第 24 章)。为了将这样的列表插入到当前行之后，可以输入：

```
:r !ls
```

作为第二个例子，下面示范如何将当前时间和日期插入到编辑缓冲区第一行的上面：

```
:Or !date
```

为了完结对:**r** 命令的讨论，下面介绍一个节省时间的好方法，该方法可以展示该命令的强大功能。在第 19 章中，我们讨论了如何使用 **look** 命令帮助查找单词的拼写。例如，假设您希望使用单词“**ascetic**”，但是不确定如何拼写这个单词。您可以使用 **look** 命令显示字典文件中可能的单词：

```
look asc
```

下面是一些典型的输出。快速地浏览一遍这个列表，就可以发现希望使用的单词位于第 5 行上：

```
ascend  
ascendant  
ascent  
ascertain  
ascetic  
ascribe
```


ascription

假设您正在使用 **vi** 向编辑缓冲区中键入文本，而此时您正好希望将这个特定单词插入到文本中。按下 <Esc> 键从输入模式切换到命令模式，然后输入命令：

```
:r !look asc
```

look 命令的输出将插入到当前行(刚刚键入的一行)之后。查看列表并删除不希望的单词。在这个例子中，除了第 5 个单词之外，将其他单词都删除(如果没有一个单词是需要的，则可以使用 **u** 命令撤销插入)。当只留下正确的单词之后，就可以将光标移动到所键入的最后一行，然后键入：

J

这样将把新单词连接到这一行的末尾。最后，为了返回到输入模式，准备在这一行的末尾插入文本，键入：

A

这样将允许在当前行的末尾追加数据。下面就是您自己的事了：继续键入数据。

起初，类似于上面的这一连串命令看上去似乎有点复杂。实际上，一旦您习惯了它，就会觉得它非常简单。此外，您的双手可以不必离开键盘完成整件事情。试一试，这是一种非常酷的体验。

:r 命令概括如下：

:liner file	在指定行后插入文件 <i>file</i> 的内容
:r file	在当前行后插入文件 <i>file</i> 的内容
:liner !program	在指定行后插入程序 <i>program</i> 的输出
:r !program	在当前行后插入程序 <i>program</i> 的输出

提示

如果对 **:r** 或 **:r!** 命令的结果不满意，可以使用 **u**(undo, 撤销)命令撤销它们。

22.37 使用程序处理数据：fmt

使用 **!** 和 **!!**(感叹号)命令将把编辑缓冲区中的行发送给另一个程序。该程序的输出将替换原始行。例如，可以用排好序的行替换原来的行。下面解释其工作方式。

将光标移动到希望处理的开始位置。键入希望处理的行数，后面跟 **!!**(两个感叹号)以及程序名称，再后跟 <Return> 键。例如，假设您有 5 行文本，包含下述数据，而您希望将它们排序(**sort** 程序在第 19 章讨论过)。

```
entertain  
balloon
```

```
anaconda
dairy
coin
```

将光标移动到第一行，输入：

```
5!!sort
```

一旦键入第二个`!`，`vi` 就会将光标移动到命令行，并显示一个`!`字符。然后您就可以直接在命令行上键入任意的 `shell` 命令。如果需要，还可以在按`<Return>`键之前使用`<Backspace>`(或`<Delete>`)键进行纠正。在我们的例子中，原始的 5 行将被替换为：

```
anaconda
balloon
coin
dairy
entertain
```

如果不喜欢这个结果，依然可以使用 `u` 命令撤销这些改变。

下面再举一个例子，这个例子也特别有用。在第 18 章中，我们讨论了 `fmt`(format, 格式化)程序。该程序重新格式化文本，使其每行不超过 75 个字符(默认)。在这样做时，`fmt` 保留行首的空格、单词之间的空格以及空白行。换句话说，`fmt` 在不改变段落结构的情况下，使文本看上去更好看。

在创建文档的过程中，`fmt` 在对编辑缓冲区全部或者部分内容进行格式化处理时非常有用。一旦知道了如何使用 `fmt`，就不必在输入或修改数据时担心换行问题，因为随时可以使用 `fmt` 处理换行问题。作为示范，下述命令从当前行开始格式化 10 行：

```
10!!fmt
```

到目前为止，所有的例子使用的都是`!!`(双感叹号)命令。`!`(单感叹号)命令的工作方式与此相似，只是`!`命令在指定输入行的范围时更具有灵活性。

键入`!`，后面跟一个光标移动命令，再后跟程序的名称。从当前行到光标移动命令指向的位置之间的所有行都将发送给程序进行处理。例如，假设希望格式化从当前行至段落末尾之间的文本。前面解释过，`}`(右花括号)命令可以将光标移动到段落的末尾，所以可以使用：

```
!)fmt
```

本章后面将示范如何使这条命令更易于使用(参见有关宏的一节)。

下面介绍一种格式化整个编辑缓冲区的简易方法。通过键入 `gg` 或 `1G` 跳转至编辑缓冲区的第一行，然后输入：

```
!Gfmt
```

(记住，`G` 命令跳转到编辑缓冲区的末尾。)

同理，可以先使用 **gg** 或 **1G** 命令，然后再使用下述命令将整个编辑缓冲区排序：

```
!Gsort
```

概括起来：

n!!program 在 *n* 个行上执行程序 *program*

!move program 从当前行至 *move* 行执行程序 *program*

22.38 将数据写入文件

当使用 **ZZ** 命令停止 **vi** 时，**vi** 会自动地将数据保存起来。但是，**vi** 中还有几条命令可以用来随时将数据写入到文件中。这些命令非常重要，因为它们允许在不退出 **vi** 的情况下，不时地备份数据。这些命令还允许将数据保存到不同的文件中。这些命令为：

:w 将数据写入原始文件

:w file 将数据写入到一个新文件中

:w! file 覆盖一个已有的文件

:w>> file 将数据追加到指定的文件中

:w 命令将编辑缓冲区中的内容写入到原始文件，替换该文件的当前内容。例如，假设您通过输入下述命令启动 **vi**：

```
vi memo
```

vi 启动后，文件 **memo** 的内容被复制到编辑缓冲区。无论对编辑缓冲区如何进行修改，原始文件 **memo** 的内容都保持不变。这非常重要，因为这样才可以在不改变原始文件的情况下退出 **vi** (使用 **:q!** 命令)。但是，在任何时候，可以使用下述命令将编辑缓冲区的内容复制到原始文件中：

```
:w
```

通常，除非准备使用 **:e** 命令编辑一个新文件 (参见下面)，否则不需要这样做。但是，如果做了大量的修改，那么您可能希望将它们保存到原始文件中。这样就可以防止由于意外事故而丢失前面的工作。

如果在 **:w** 命令之后指定一个文件的名称，那么 **vi** 将把编辑缓冲区中的数据写入到这个文件中。例如，为了将编辑缓冲区中的内容保存到文件 **extra** 中，可以输入：

```
:w extra
```

如果这个文件不存在，那么 **vi** 会创建这个文件。如果这个文件已经存在，那么 **vi** 将显示一个警告消息。下面是两个典型的消息：

```
File exists - use "w! extra" to overwrite
```

File exists (add ! to override)

如果确实希望覆盖这个文件，则必须使用:w!命令：

```
:w! extra
```

为了将数据追加到已有文件的末尾，可以在命令名称之后键入>>(两个大于号)。例如：

```
:w>> extra
```

使用>>将保留旧数据。注意，>>表示法也用于向已有文件中追加标准输出(参见第15章)。

如果只希望保存编辑缓冲区中的特定行，则可以采用通常的方法指定行号。例如，为了将第10行写入到文件save中(替换save的内容)，可以输入：

```
:10w! save
```

为了将第10行至第20行追加到文件save中，可以使用：

```
:10,20w>> save
```

在本章前面，我们解释过有两种方法可以退出vi。要保存数据并退出，可以使用命令ZZ；要在不保存数据的情况下退出vi，可以使用:q!。实际上，还有第三种方法。只要已经使用:w命令保存了数据，就可以使用:q退出vi。为了方便起见，还可以组合使用这两条命令：

```
:wq
```

因此，组合命令:wq和ZZ拥有相同的效果。

22.39 切换到一个新文件

当启动vi时，可以指定希望编辑的文件的名称。例如，为了编辑文件memo，可以输入：

```
vi memo
```

如果在编辑一个文件的时候，决定编辑另一个文件，这时不必退出并重启vi程序。为了切换到一个新文件，可以使用:e命令，后面跟新文件的名称。例如，为了编辑文件document，可以输入：

```
:e document
```

当开始编辑一个新文件时，编辑缓冲区中的原有内容将会丢失，所以要确保首先使用:w命令将数据保存。当使用:e命令时，vi将检查数据是否已经保存。如果还有未保存的数据，那么vi将不允许切换到新文件。如果希望忽略这种保护，可以使用:e!命令。例如，

假设使用下述命令启动 vi:

```
vi memo
```

memo 的内容被复制到编辑缓冲区中。但是，由于在编辑过程中犯了许多错误，因此您决定重新打开这个文件。为了忽略编辑缓冲区中数据的改变，重新打开原始文件，可以输入:

```
:e!
```

现在就可以对原始文件的副本进行编辑了——以前的修改都已被丢弃。概括起来:

```
:e file 编辑指定的文件
```

```
:e! 重新编辑当前文件，忽略自动检查
```

```
:e! file 编辑指定的文件，忽略自动检查
```

22.40 使用缩写

要为经常使用的单词或表达式创建缩写，可以使用 **:ab**(abbreviate, 缩写)命令。该命令的语法为:

```
:ab [short long]
```

其中 *short* 是缩写，*long* 是缩写的原文。

下面举例说明。假设您正在为暑假的实习工作准备一份履历表，需要不停地键入“exceptionally gifted”，这让人觉得很烦。因此，您希望建立一个缩写，比如“eg”。键入 **:ab**，后面跟着短格式的缩写，再后跟长格式的原文:

```
:ab eg exceptionally gifted
```

从现在开始，在位于输入模式时，每当以一个单独的单词键入 **eg**，**vi** 都将自动地使用“exceptionally gifted”替换它。注意只有“eg”是一个单独的单词时才会进行替换，**vi** 非常聪明，不会替换其他单词中的“eg”，例如“egotistical”。

如果要删除一个缩写，可以使用 **:una**(un-abbreviate, 反缩写)命令。该命令的语法为:

```
:una short
```

其中 *short* 是缩写。在使用这个命令时，只需键入 **:una**，后面跟着希望移除的短格式的缩写。例如:

```
:una eg
```

在任何时候，通过输入 **:ab** 命令本身可以查看所有的缩写列表:

```
:ab
```

提示

如果要在每次启动 **vi** 时自动地定义缩写, 可以在初始化文件中放入合适的 **:ab** 命令(本章后面讨论)。

22.41 宏

正如上一节中讨论的, **:ab** 命令可以用来创建输入模式中使用的缩写。这使您不必一遍又一遍地键入相同的文本, 而只需使用一个缩写。类似地, 可以用 **:map** 命令创建命令模式中使用的单字符的缩写。实际上, 这将允许创建自己定制的单字符命令, 这些命令称为宏(macro)。该命令的语法为:

```
:map [x commands]
```

其中 *x* 是一个字符, *commands* 是 **vi** 或 **ex** 命令序列。

下面是一个简单的宏。在本章前面(在有关移动文本的那一节中), 我们展示了如何调换两个单词: 将光标移动到第一个单词前面的空格处, 然后键入 **deep**。为了使这个操作更加简便, 可以定义这样一个宏:

```
:map K deep
```

现在, 为了调换两个单词, 只需将光标移动到第一个单词前面的空格处, 按 **K** 键即可。

根据定义, 宏的名称必须是单个字符。如果使用一个早已拥有其他含义的字符, 那么这个字符将会失去原本含义。例如, 在本章前面, 我们讨论了 **x**(小写字母“x”)和 **X**(大写字母“X”)命令。**x** 命令删除当前字符(光标指向的字符); **X** 命令删除光标左边的字符。考虑下述宏定义:

```
:map X dd
```

该命令创建一个命名为 **X**(大写字母“X”)的宏, 这个宏删除当前行。一旦定义了这个宏, 常规的 **X** 命令将不起作用。然而, 这一点并不用太介意。如果您极少使用该命令, 那么您可能发现通过键入 **X** 删除一行更有意义。

但是, 替换常规的命令通常并不是一个好主意。这将产生一个问题: **vi** 或 **Vim** 还没有使用的是哪些字符呢? 实际上, 没有使用的字符极少。如图 22-10 所示, **vi** 没有使用的字符只有 14 个。**Vim** 更加极端: 没有用作命令名称的常见字符只有 **^K** 和 ****(反斜线)。不过, 情形并没有想象的那么糟糕, 因为有若干条 **Vim** 命令或许根本不需要, 可以安全地替换它们。这些字符如图 22-10 所示。

vi: 可以用作宏名称的字符	
字母	g K q v V Z
标点符号	@ # * \
Ctrl 字符	^A ^K ^O ^W ^X

Vim: 可以用作宏名称的字符	
字母	K q v V
标点符号	@ \
Ctrl 字符	^@ ^A ^K ^O ^T ^X

图 22-10 用作 vi 和 Vim 宏名称的字符

:map 命令可以创建宏，即 **vi** 或 **ex** 命令序列的缩写。宏名称必须是单个字符。如果选择早已拥有含义的名称作为命令，那么原有含义将会丢失。第一个表列举的字母是那些 **vi** 命令还没有使用的字符，因此，可以安全地用作宏的名称。

对于 Vim 来说，几乎所有的字符都用作命令名称。然而，一些命令并不经常使用，所以可以替换它们。这些字符列举在第二个表中。

从真实意义上讲，宏就是微型程序。对于所有的编程工具来说，有许多细节可能永远都不需要，因此这里不讨论全部细节^{*}。作为替代，我们将示范几个比较有用的宏，以展示宏的用途。

正如本章前面讨论的，**G** 命令将光标移动到编辑缓冲区的末尾。**1G**(跳转到第 1 行)命令将光标移动到编辑缓冲区的开头。对于一些版本的 **vi**(以及 Vim)来说，可以使用 **gg** 替换 **1G**。但是，如果您使用的 **vi** 不支持 **gg**，那么定义一条简单的命令替换它就比较方便。考虑下述命令：

```
:map g 1G
```

该命令定义了一个代表命令 **1G** 的宏 **g**。现在，当希望在编辑缓冲区中自由跳转时，可以键入 **g**(小写字母“g”)跳转到缓冲区的开头，或者键入 **G**(大写字母“G”)跳转到缓冲区的末尾。

下面再示范一个更复杂的宏。假设您正在使用诸如 C 或 C++之类的语言编写一个程序，在这类语言中注释用 **/***和***/**括起来，例如：

```
/* This line is a comment */
```

下面的宏通过在普通行的行头插入 **/***，在行尾插入 ***/**创建一个注释：

^{*} 如果您学会了足够的技巧，并且有足够的时间，那么您可以编写非常复杂的 **vi** 宏。例如，有些人编写宏来解决汉诺塔问题，或者模拟图灵机(两个经典的计算机科学问题)。如果您喜欢这类事情，则应该知道 Vim 要比标准 **vi** 提供更多复杂的工具。对于 Vim 来说，宏可以被记录、重放及修改。另外，还可以使用成熟的脚本工具编写程序。更多的信息，请在 Web 页面上搜索“vim macros”和“vim scripting”。

```
:map * I/* ^V<Esc>A */^V<Esc>
```

下面我们剖析这个宏。在`:map`之后是一个`*`(星号)字符, 这就是宏的名称。

接下来的是命令。首先, 我们使用 `I` 在行的开头进入输入模式。然后, 键入`/*`, 后面跟一个空格。此时, 我们需要按`<Esc>`键退出输入模式。为了在宏中插入一个`<Esc>`码, 我们键入`^V<Esc>`(正如本章前面讨论的, `^V[Ctrl-V]`告诉 `vi` 接下来的键要从字面上解释)。

接下来, 我们使用 `A`(append, 追加)命令在行的末尾进入输入模式。然后, 键入一个空格, 后面跟着`*/`。为了退出输入模式, 我们再次使用了`<Esc>`键。

如果在命令行上输入上述命令, 可以发现`<Esc>`会显示为`^[]`。换句话说, 在屏幕上看到的内容是:

```
:map * I/* ^[]A */^[]
```

这是因为`<Esc>`码实际上就是`^[]`, 同样, 退格键是`^H`(参见第 7 章), 制表符是`^I`。

一些版本的 `vi` 允许将宏赋给`<F1>`到`<F10>`的功能键。这时, 可以通过在一个`#`字符之后跟一个数字(`1=F1`、`2=F2`……`0=F10`)来指定自己喜欢的功能键。例如, 下述命令创建一个宏, 并赋给`<F1>`键:

```
:map #1 :set all
```

在任何时候都可以使用`:map`命令本身显示所有宏的列表:

```
:map
```

在移除宏时, 可以使用`:unmap`命令。该命令的语法为:

```
:unmap x
```

其中 `x` 是宏的名称。例如:

```
:unmap g
:unmap #1
```

22.42 初始化文件: `.exrc`、`.vimrc`

在 `vi` 或 `Vim` 启动时, 它会在 `home` 目录中查找初始化文件。如果存在这样的文件, 程序将读取它并执行任何查找到的 `ex` 命令。通过这种方式, 可以自动地初始化工作环境(`home` 目录在第 23 章中讨论, 初始化文件在第 14 章中讨论)。

对于 `vi` 来说, 初始化文件名为`.exrc`(发音为 dot-E-X-R-C)。对于 `Vim` 来说, 初始化文件名为`.vimrc`(发音为 dot-vim-R-C)*。正如第 14 章讨论的, 开头的`.`(点号)表示这些文件是隐藏文件; “rc” 标记则代表 “run commands, 运行命令”。

* `Vim` 首先查找`.vimrc`。如果这个文件不存在, 那么 `Vim` 接着查找`.exrc` 文件。因此, 如果同时有这两个文件, 那么 `Vim` 将只读取`.vimrc` 文件。

创建 vi 初始化文件非常简单：只需插入希望每次启动 vi 时自动执行的 **ex** 命令即可。具体而言，应该包括所有经常使用的 **:set**(选项)、**:ab**(缩写)以及 **:map**(宏)命令。另外还可以使用 **:!命令**来运行 shell 命令。

在 vi 读取初始化文件时，以“(双引号)字符开头的各行都被忽略，这意味着可以使用这样的行进行注释。同样，行头的空格和制表符也都被忽略，从而允许出于可读性考虑对行进行缩进。^{*}最后，vi 假定它读取的所有内容都是 **ex** 命令，因此命令不需要以冒号开头。

为了描述这些思想，图 22-11 中示范了一个可以在 vi 或 Vim 中使用的初始化文件样本。

```
" =====
" Sample vi/Vim initialization file
" =====
"
" 1. Options
"   set autoindent
"   set compatible
"   set ignorecase
"   set showmatch
"   set showmode
"   set wrapmargin=6
"
" 2. Abbreviations
"   ab eg exceptionally gifted
"   ab h Harley
"
" 3. Macros
"   map K deep
"   map X dd
"   map g lG
"   map #5 {!}fmt^M
"
" 4. Shell commands
"   !date; sleep 2
```

图 22-11 vi/Vim 的初始化文件样本

当启动 vi 或 Vim 时，程序将在 home 目录中查找初始化文件。这里是一个可以用作初始化文件模板的样本文件。详情请参见正文。

第一节设置如下选项。

autoindent: (输入模式)当对自动缩进使用 **wrapmargin** 时，匹配上一行或下一行的缩进。

compatible: (只适用于 Vim)强制 Vim 以 vi 兼容模式运行。如果 Vim 没有以 vi 模式运行，则使用该选项，即便使用了 **-C** 选项。

exrc: 当启动时，在当前目录中查找第二个初始化文件(下一节中解释)。

ignorecase: 当搜索时，忽略大写字母和小写字母之间的区别(非常便利)。

showmatch: (输入模式)当键入右圆括号、方括号或者花括号时，高亮显示匹配的左圆括号、方括号或花括号。

^{*} 一些版本的 vi(以及 Vim)还忽略初始化文件中的空白行，这可以让您加入更多空行，以使文件更易于阅读。如果您的 vi 不允许空白行，而您使用了空白行，那么您将看到一个不明确的错误消息，例如“Error detected in .exrc”。

showmode: 位于输入模式时显示一个提醒。

wrapmargin: 指定文本接近到行末尾什么程度时触发行的自动缩进。

第二节创建缩写。这里是为不易处理的单词、HTML 标签、编程关键字等指定快捷键的好位置。

第三节定义宏。正如本章前面所述，宏可以非常复杂(特别是在 Vim 中)。这里的样本宏非常简单。但是，我希望强调两点。

首先，如果 **vi** 支持 **gg** 命令，则不需要宏 **g**。

其次，使用功能键<F5>(#5)的宏以[^]**M** 字符(回车代码)结束，来模拟<Return>键的按键。注意这是一个控制字符，不是两个单独的字符。当在初始化文件中键入这一行时，必须按[^]**V**<Return>或者[^]**V**[^]**M** 键来插入一个实际的[^]**M**。

第四节包含 **shell** 命令。我在这里只使用了一条命令来示范它的使用方式。在这个例子中，我使用的是 **date** 命令(参见第 8 章)来显示时间和日期。**sleep** 命令用来暂停指定的时间(这个例子中为 2 秒)。

如果初始化文件中包含有错误命令，那么 **vi** 将显示一个错误消息，并从这个地方退出正在执行的文件。尽管 **vi** 将正常启动，但是初始化文件中的其他初始化命令将得不到执行。而 Vim 就比较厚道：它显示一个错误消息，要求您按<Enter>键确认，然后继续执行初始化文件中的下一条命令。

22.43 使用两个初始化文件

对于需要额外定制的情形，可以使用一个额外的初始化文件。在解释其工作方式之前，先简要地提一下将在第 24 章中讨论的两个思想。

当创建用户标识时，用户标识获得自己的目录，称为“home 目录”。在 home 目录中，可以根据需要创建任意数量的子目录。每次登录时，都是从 home 目录开始工作的。但是，在工作时也可以方便地从一个目录切换到另一个目录。在任何时候，当前正在使用的目录称为“工作目录”。

当运行 **vi** 或 Vim 时，它们通过执行 home 目录中的初始化文件完成启动。然后，检查 **exrc** 选项的状态。如果该选项是打开的，那么程序将在当前目录中查找第二个初始化文件来执行(假定当前目录与 home 目录不同)。通过这种方式，可以将各个文件组织到子目录中，每个子目录都有自己的初始化文件。

例如，假设您当前正在处理 3 个项目：一篇论文、一个程序以及一个 Web 页面。这些项目的文件分别保存在 3 个不同的目录 **essay**、**program** 和 **webpage** 中。在每个目录中都创建了一个自定义的 **.exrc** 或 **.vimrc** 文件，这些文件中包含有编辑特定项目的文件时希望使用的选项、缩写和宏。

假定您现在处于 **program** 目录中，希望编辑文件 **test.c**。您输入命令：

```
vi test.c
```

一旦 **vi** 启动，它就在您的 **home** 目录中查找初始化文件。执行了初始化文件中的命令后，**vi** 将查看 **exrc** 选项。如果这个选项是打开的，那么 **vi** 就在 **program** 目录(当前目录)中查找第二个初始化文件。通过这种方式，就能够以适合编辑程序的方式定制工作环境。

22.44 学习使用 Vim

在本章的开头，我们讨论了 Vim，一种向后兼容 **vi** 的复杂文本编辑器。Vim 由荷兰程序员 Bram Moolenaar 于 1988 年创建，是 **vi** 克隆版本的“改进”版。从那时起，Vim 开始急剧地扩展，增加了数百种新特性。同时，Vim 逐渐流行起来，在许多 Unix 和 Linux 系统上取代了 **vi**。尽管可以以“**vi** 兼容”模式运行 Vim，但是它并不仅仅是一种 **vi** 的改进版本。Vim 本身是一种非常复杂的编辑器，与 **vi** 存在着显著的不同。

事实真相是，Vim 是如此的复杂，以至于不可能有人能教会您使用 Vim。您必须自学如何使用 Vim。但是，当您刚开始时，就会立即发现一个问题：Vim 的文档资料并不适合于初学者。解决方法就是从学习 **vi** 开始。一旦理解了 **vi**，就会拥有理解 Vim 的基础，然后才可以一点一点地自学 Vim。这就是本章主要讨论 **vi**，而不是 Vim 的原因之一(另一原因是 **vi** 到处存在，而 Vim 在许多 Unix 系统上不可用)。

因此，如果您希望学习 Vim，下面是我的建议。首先阅读本章并进行练习，直至感觉自己掌握了 **vi**。这至少要一两个月的时间。在这段时间内，应该以 **vi** 兼容模式运行 Vim(这样做的指令参见本章前面)。然后，可以关闭兼容模式，按照 Vim 的原本面貌运行 Vim。

我猜测您已经使用过许多程序——特别是基于 GUI 的程序，在使用它们的过程中就可以快速地熟习它们。但是，Vim 有所不同。Vim 是一种必须自学的程序，在自学过程中，必须阅读文档资料。^{*}首先在 shell 提示处运行下述命令：

```
vimtutor
```

这将显示一个指南，汇总了 Vim 的基本命令(大多数命令已经在本章中学习过)。当阅读完后，可以键入 **:q** 退出。

接下来，启动 Vim，依次输入下述命令浏览联机帮助(在退出时还是键入 **:q**)。

```
:help  
:help user-manual  
:help differences
```

如果发现通过这种方式阅读文档资料单调乏味(和我的感觉一样)，则可以在线阅读。这些资料位于 **www.vim.org** 上(在这个网站上查找 Vim 的“User Manual”)。

请不要被我的建议吓倒。Vim 是一个令人吃惊的程序，如果您觉得它非常有吸引力，那么请一定要学习它。为了激发您的学习激情，图 22-12 中列举了 Vim 提供的最重要的增强。在基于 **vi** 的知识学习 Vim 的过程中，我的建议就是按图中所示的顺序学习如何使用各

^{*} 如果请求其他人教授您如何使用 Vim，那么您只会对他所讲的内容感到困惑，他也会觉得失望。不要说我没有警告过您。

种特性。

Vim 是一个高度完善的文本编辑器，相对于标准 vi，它提供了许多增强特性。下面是 Vim 提供的最重要的增强特性一览表。学习 Vim 的最佳方式就是首先掌握 vi，然后再按下面列举的顺序自学如何使用 Vim 的增强特性。

- 广泛的联机帮助。
- 屏幕拆分：可以水平或垂直拆分屏幕，每个窗口都可以显示自己的文件。
- 多级撤消。
- 支持鼠标。
- 支持 GUI。
- 命令行历史。
- 命令行补全。
- 文件名补全。
- 搜索历史。
- 语法高亮显示：使用颜色显示众多不同类型文件的语法。
- 高亮显示：选取文本行或者文本块，然后再对文本进行操作。
- 多缓冲区。
- 支持宏：记录、修改和运行宏的工具。
- 内置脚本语言：创建自己的脚本，分享他人编写的脚本(Internet 上免费可用)。
- 自动命令：自动执行预定义命令。

图 22-12 Vim：相对于标准 vi 的增强

22.45 事实背后的故事

在结束本章时，我希望告诉大家一个真实的故事，这个故事展示了美国诗人 John Greenleaf Whittier(1807–1892)在他的诗 *Maud Muller* 中表达的那种渴望，他写道：

For of all sad words of tongue or pen,

The saddest are these: "It might have been!"

从图 22-12 中可以看出，Vim 提供的重要增强之一就是“屏幕拆分”，即将屏幕拆分成水平窗口或垂直窗口。这是一个功能强大的工具，允许同时查看多个文件。

更有趣的是，追溯到 1978 年的加利福尼亚大学伯克利分校，Bill Joy(vi 的创建者)正计划将这一特性添加到该程序的早期版本中。下面是他自己的话，来自 *Unix Review* 杂志 1984 年的一次访谈：

“实际上，在我们安装 VAX(计算机)时，我正准备在 vi 中添加多窗口特性，这一时间是 1978 年 12 月。我们没有任何的备份，而磁带驱动器也坏了。即便没法备份，我也一直持续工作——但是，后来源代码被破坏了，而我连一份完整的清单都没有。”

“我几乎快要重写完全部的窗口显示代码，但是这时我放弃了。在这之后，我又回到以前的版本，只是记录代码、补充手册，使其更加完善。如果源代码不被破坏的话，或许 vi 早已拥有了多窗口特性，而且我还有可能在其中添加一些编程特性，但现在这都已经不好再说了。”

碰巧的是，那个时候，美国哲学家 Roseanne Roseannadanna 正好访问伯克利的 Unix 实验室。当 Joy 告诉她所发生的事情时，她评论道：“很好，Bill，它就像我的父亲曾经对我说的，‘如果它不是一件事情，那么它就是另一件事情’。”

22.46 练习

1. 复习题

1. 当希望完成下述任务时，如何启动 **vi**：编辑文件 **document**？编辑一个全新的文件？以只读模式打开文件 **document**？以 **vi** 兼容模式启动 Vim？

2. 如果已经保存了工作，那么如何退出 **vi** 呢？如何保存工作然后退出？如何不保存工作而退出？

3. 在使用 **vi** 时，数据保存在一块存储区中。这块存储区叫什么？**vi** 编辑器以两种主要模式进行操作：命令模式和输入模式。描述每一种模式。如何由命令模式切换到输入模式？如何由输入模式切换到命令模式？

4. 找出将光标移动到下述指定位置的最佳命令，尽可能地使用字母键：

- 左、下、上、右移一个位置
- 当前行的开头
- 当前行的末尾
- 上一行的开头
- 下一行的开头
- 向前移动一个单词
- 向后移动一个单词
- 向前移动到下一个句子
- 向后移动到上一个句子
- 向前移动到下一个段落
- 向后移动到上一个段落
- 屏幕的最顶行
- 屏幕的中间行
- 屏幕的最底行
- 编辑缓冲区的开头
- 编辑缓冲区的末尾
- 下移一屏幕
- 上移一屏幕
- 下移半屏幕
- 上移半屏幕

5. 在命令模式中，如何完成下述操作：

- 撤销修改编辑缓冲区的上一条命令
- 将当前行恢复到刚刚移动到它时的内容
- 重复执行修改编辑缓冲区的上一条命令

2. 应用题

1. 启动 **vi** 并创建一个全新的空文件 **temp**。将下述行插入到文件中：

```
one 1
two 2
three 3
four 4
five 5
```

使用一条单独的 **vi** 命令保存工作并退出 **vi**。

2. 启动 **vi** 编辑前一个问题中的文件 **temp**。

只使用 **vi** 命令：将第 2 行至第 4 行移动到第 5 行之后；撤销移动。

只使用 **vi** 命令：复制第 2 行至第 4 行到编辑缓冲区的顶部；撤销改变。此时，编辑缓冲区看上去应该与刚启动时相同。不保存退出。

3. 启动 **vi** 编辑习题 1 中的文件 **temp**。

尽可能使用 **ex** 命令：将第 2 行至第 4 行移动到第 5 行之后；撤销改变。

尽可能使用 **ex** 命令：复制第 2 行至第 4 行到编辑缓冲区的顶部；撤销改变。此时，编辑缓冲区看上去应该与刚启动时相同。不保存退出。

比较习题 2 中使用的 **vi** 命令与习题 3 中使用的 **ex** 命令。**ex** 命令有什么优点呢？

4. 启动 **vi** 编辑习题 1 中的文件 **temp**。在编辑缓冲区的底部插入时间和日期。光标在什么位置？为什么？

不移动光标，使用一条命令对编辑缓冲区的所有行按字母表的逆序进行排列。不保存退出 **vi**。

3. 思考题

1. 一旦习惯了使用 **vi** 编辑器，就会发现 **vi** 编辑器其实是一个快速、强大而且易于使用的编辑器。但是，**vi** 是一个非常复杂的程序，需要花费大量的精力来掌握。向后兼容的替代程序 **Vim** 功能更强大，但是也更复杂，学习更加困难。考虑到 **vi** 已经有 30 多年的历史，而且非常难以学习，您认为 **vi** 为什么在 Unix 社区中如此流行呢？

您认为未来复杂的任务将使用专有的易于使用的工具来执行，还是依旧使用诸如 **vi** 之类的程序呢？

2. 广义地讲，**vi** 有两种不同类型的命令：面向屏幕的 **vi** 命令和面向行的 **ex** 命令。实际上，这两种类型的命令完全不同，是为不同类型的硬件开发的。然而，它们组合在一起，创建了一个功能强大的编辑环境。这是为什么呢？这说明我们应该为聪明人设计什么类型的工具呢？

Unix 文件系统

在接下来的 3 章中，我们将讨论 Unix 文件系统，即操作系统中通过存储及组织系统中的全部数据而为用户和程序提供服务的那一部分。在本章中，我们将讨论一些基本的概念。然后，在第 24 章中讨论使用目录的细节，在第 25 章中讨论使用文件的细节。

本章的目的是回答 3 个关键问题。第一，什么是 Unix 文件？可以想象，文件是数据在磁盘上的储存库。但是，可以看出，文件的含义不仅仅限于这一点。第二个问题针对的是文件的组织。通常，一个 Unix 系统拥有数十万个文件。如此众多的对象以何种方式组织才有意义，才易于理解呢？最后一个问题就是，一个独立的系统有没有可能为众多不同类型的数据存储设备提供透明的支持？

我们的讨论从一个简单但拥有令人惊讶的复杂答案的问题开始，这个问题就是：什么是文件？

23.1 什么是文件

以前，在计算机发明之前，术语“文件”指的是一组纸张。一般情况下，文件存放在纸板文件夹中，而文件夹在文件柜中组织和存放。现在，大多数数据都是计算机化的，因此文件的定义也发生了相应的变化。从最简单的意义上讲，文件就是一个有名称的数据集合^{*}。大多数时候，文件存储在数字介质上：硬盘、CD、DVD、软盘、闪存、存储卡等。

在 Unix 中，文件的定义更为广泛些。文件是任意源，有一个名称，可以从中读取数据；或者是任意目标，有一个名称，可以向其中写入数据。因此，当使用 Unix 或 Linux 时，术语“文件”不仅指像磁盘文件那样的数据储存库，而且还指任意的物理设备。例如，键盘(一种输入源)、显示器(一种输出目标)都可以作为文件被访问。另外还有不存在所谓的物理实体的文件，它们也接收输入或产生输出，从而提供具体的服务。

以这种方式(具有广泛的通用性)定义文件拥有重大意义：它意味着 Unix 程序可以使用简单的过程从任意的输入源读取数据，或者向任意的输出目标写入数据。例如，大多数 Unix 程序都设计为从标准输入读取数据，向标准输出写入数据(参见第 18 章和第 19 章)。从程

^{*} Unix 文件实际上拥有不止一个名称。我们将在第 25 章讨论链接时介绍这一思想。

程序员的角度来看, I/O(输入/输出)非常简单, 因为数据的读取和写入可以以一个简单标准的方式来实现, 无论实际数据来源于何处或者发送到何方。从用户的观点来看, 这提供了极大的灵活性, 因为用户可以在运行程序时指定输入源和输出目标。

可以想象, Unix 文件系统(或者任意其他文件系统)的内部细节都非常复杂。在本章中, 我们将讨论一些基本的概念。当阅读完本章内容后, 您就会发现 Unix 文件系统具有一些引人注目的美妙之处: 这种类型的美只能在那些复杂, 但是每件事物都有意义的系统中发现。

23.2 文件类型

Unix 文件有许多不同的类型, 但是它们都可以分成 3 类: 普通文件、目录和伪文件。在伪文件中, 有 3 种特定的类型最常见, 它们是: 特殊文件、命名管道及 `proc` 文件。在本节中, 我们将快速地浏览一遍最重要的文件类型。在以后各节中, 将详细地讨论每一种文件类型。

普通文件(ordinary file)或常规文件(regular file)是大多数人在使用单词“文件”时所指的文件。普通文件包含数据, 位于某种类型的存储设备上, 例如磁盘、CD、DVD、闪存、存储卡或者软盘。就这一点而论, 普通文件就是大多数时间所使用的文件类型。例如, 当使用文本编辑器编写 `shell` 脚本时, `shell` 脚本和编辑器程序本身都以普通文件存储。

正如第 19 章中讨论的, 广义地讲, 普通文件有两种类型: 文本文件和二进制文件。文本文件包含的数据行由可显示的字符(字母、数字、标点符号、空格、制表符)和每行末尾的新行字符构成。文本文件用于存储文本数据: 纯文本、`shell` 脚本、源程序、配置文件、HTML 文件等。

二进制文件包含非文本数据, 这种类型的数据只有在执行或者由其他程序解释时才有意义。常见的二进制文件包括可执行程序、对象文件、图像、音乐文件、视频文件、字处理文档、电子表格、数据库等。例如, 文本编辑器程序就是一个二进制文件, 而用文本编辑器编辑的文件是文本文件。

因为实践中遇到的文件几乎全部都是普通文件, 所以学习基本的文件操作技能非常关键。具体而言, 就是必须学习如何创建、复制、移动、重命名以及删除这样的文件。我们将在第 25 章中详细讨论这些主题。

第二种类型的文件是**目录**。和普通文件相似, 目录也驻留在某种类型的存储设备上。但是, 目录不存放常规数据。它们用来组织、访问其他文件。从概念上讲, 目录“包含”其他文件。例如, 您可能拥有一个名叫 `vacation` 的目录, 其中存放着与去 Syldavia 旅行有关的所有文件。

目录中还可以包含其他目录。这样就可以按照层次结构系统组织文件。在本章后面可以看出, 整个 Unix 文件系统组织成一个大的层次结构树, 目录中套着目录, 被包含的目录中又套着目录。在自己的那一部分树中, 可以根据需要创建或删除目录。通过这种方式, 可以按照自己的意愿组织文件, 并且在需要改变时进行改变。

(在第 9 章讨论 Info 系统时我们就谈到过树。正式地讲, 树是一种数据结构, 由一组节点、叶子和分支构成, 在树的组织结构中, 任意两个节点之间至多一条分支。)

有时候可能看到使用的是术语**文件夹**，而不是“目录”，特别是在使用 GUI 工具时。这个术语来源于 Windows 和 Macintosh 领域，这两种系统都使用文件夹组织文件。Windows 的文件夹与 Unix 的目录有许多相似之处，但是没有那么强大。Macintosh 的文件夹就是 Unix 目录，因为 Mac 操作系统 OS X 就运行在 Unix 之上(参见第 2 章)。

最后一种文件类型是**伪文件(pseudo file)**。与普通文件和目录不同，伪文件并不用来存储数据。基于这一原因，这些文件本身不占用任何空间，尽管它们也被认为是文件系统的一部分，并且按目录进行组织。伪文件的目的是提供一种服务，这种服务采取和常规文件相同的访问方式进行访问。在大多数情况下，伪文件用来访问内核(操作系统的核心部分，参见第 2 章)提供的服务。

最重要的伪文件类型是**特殊文件**，有时候也称为**设备文件**。特殊文件是物理设备的内部表示。例如，键盘、显示器、打印机、磁盘驱动器(实际上，包括计算机或者网络中的每个设备)都可以当作特殊文件来访问。

下一个伪文件类型是**命名管道**。命名管道是第 15 章中讨论的管道功能的一个扩展，它能够将一个程序的输出连接到另一个程序的输入上。

最后一种伪文件类型是 **proc 文件**，它允许访问内核中的信息。在少数几种特定情况下，甚至还可以使用 **proc** 文件修改内核中的数据(很明显，这应该由那些水平非常高的人进行)。最初，创建这些文件的目的是用来提供正在运行的进程的信息，因此命名为“**proc**”。

名称含义

文件

当看到或听到“文件”这个单词时，必须根据上下文来理解它的含义。有时候，它指任意一种类型的文件；有时候，它只指包含数据的文件，即普通文件。

例如，假设您阅读下面这个句子：“**ls** 程序列举目录中所有文件的名称。”在这种情况下，单词“文件”指的就是任何类型的文件：普通文件、目录、特殊文件、命名管道或者虚拟文件。这 5 类文件都可以存储在一个目录中，因此，可以使用 **ls** 程序列举这些文件。

但是，假设您读到这样一句话：“为了比较两个文件，可以使用 **cmp** 程序。”在这种情况下，“文件”指的是普通文件，因为普通文件是包含可以比较的数据的唯一文件类型。

23.3 目录和子目录

我们使用目录将文件组织成一个类似于树的层次系统。具体组织时，将文件组合成组，再将每组文件存储在各自的目录中。由于目录本身就是文件，所以目录还可以包含其他目录，从而形成层次结构。

下面举例说明。假设您是某著名的西海岸大学的一名学生，您正在学习 3 门课程——历史、文学和冲浪，每一门课程都要写几篇论文。为了组织所有的工作，您创建了一个 **essays** 目录(现在还不用考虑细节问题)。在这个目录中，又创建了 3 个目录 **history**、**literature** 和 **surfing** 来存放论文。每篇论文都存储在一个拥有描述性名称的文本文件中。图 23-1 给出了这些目录和文件的一张示意图。注意这张图看上去就像一棵顶末倒置的树。

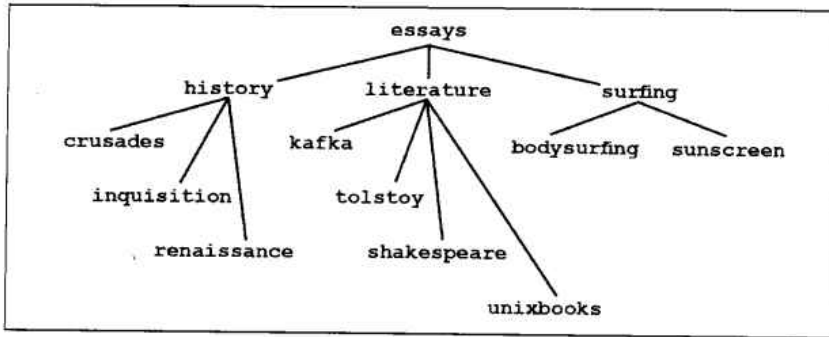


图 23-1 目录组织形式示例

为了组织文件，我们使用父目录和子目录创建一个层次结构树。在这个例子中，父目录就是 **essays**，它包含 3 个子目录 **history**、**literature** 和 **surfing**，每个子目录中又包含有普通文件。详情请参见正文。

父目录(parent directory)是包含其他目录的目录。子目录(subdirectory 或 child directory)是位于另一个目录中的目录。在图 23-1 中，**essays** 是父目录，它包含 3 个子目录 **history**、**literature** 和 **surfing**。

通常在谈论目录时就好像目录中实际包含其他文件一样。例如，我们可以说 **essays** 包含 3 个子目录，**literature** 包含 4 个普通文件。实际上，可以想象，如果查看 **literature** 目录内部，则可以看到 4 个文件。实际上，所有的文件都存储为一个单独的实体。目录并不存放实际文件。它只是包含 Unix 定位文件所需的信息。

但是，这些细节并不用去考虑，因为 Unix 自动维护着整个文件系统的内部工作。我们只需学习如何使用合适的程序，而 Unix 将完成相应的任务：生成目录、移除目录、移动目录和列举目录中的内容等。我们将在第 24 章中讨论这些程序。

23.4 特殊文件

特殊文件是表示物理设备的伪文件。Unix 将所有的特殊文件存放在 **/dev(device, 设备)** 目录中(我们将在本章后面讨论名称开头的斜线)。为了显示系统上特殊文件的名称，可以使用 **ls** 程序(参见第 24 章)：

```
ls /dev | less
```

您将看到许多名称，其中大多数名称极少需要使用。这是因为大部分特殊文件是由系统程序使用的，而不是由用户使用的。然而，有少数几个特殊文件研究起来也是很有趣的。这些文件列举在图 23-2 中，我们将把这些文件分成 3 组进行讨论：硬件、终端和伪设备。

硬件	
<code>/dev/fd0</code>	软盘
<code>/dev/hda</code>	硬盘
<code>/dev/hda1</code>	硬盘：第 1 分区
<code>/dev/sda</code>	SCSI 硬盘
<code>/dev/sda1</code>	SCSI 硬盘：第 1 分区
<code>/dev/sda1</code>	USB 闪存卡(参见正文)
<code>/dev/lp0</code>	打印机
<code>/dev/usb/lp0</code>	USB 打印机
终端	
<code>/dev/tty</code>	当前终端
<code>/dev/tty1</code>	控制台/虚拟控制台
<code>/dev/pts/0</code>	伪终端
<code>/dev/ttyp0</code>	伪终端
伪设备	
<code>/dev/null</code>	放弃输出，输入不返回内容(eof)
<code>/dev/zero</code>	放弃输出，输入返回 <code>null(0)</code>
<code>/dev/random</code>	随机数生成器
<code>/dev/urandom</code>	随机数生成器

图 23-2 最有趣的特殊文件

特殊文件是用来表示设备的伪文件。这样的文件主要由系统程序使用。尽管很少直接使用特殊文件，但是有一些有趣的特殊文件还是需要知道的。详情请参见正文。

如果您热衷于理解其他特殊文件名称的含义，那么有一个正式的列表可以提供帮助。为了查找该列表，可以在网络上搜索“LANANA Linux 设备列表”(LANANA 代表“Linux Assigned Names and Numbers Authority”)。显而易见，该列表对 Linux 特别有用。但是，大多数重要的特殊文件在所有 Unix 系统上拥有相似的名称，因此即便您不使用 Linux，这个列表也是有用的。

23.5 硬件特殊文件

所有连接到计算机上的设备都通过特殊文件访问。我们从最简单直接的设备入手，即那些表示实际硬件的设备。从图 23-2 中可以看出，文件 `/dev/fd0` 表示一个软盘驱动器。设备名称后面的数字指的是一个具体的设备。在这个例子中，`/dev/fd0` 指的是第一个软盘驱动器(计算机程序通常从 0 开始计数)。如果存在第二个软盘驱动器，那么它将是 `/dev/fd1`，依此类推。同理，`/dev/lp0` 对应于第一台打印机。

硬盘的处理有点不同。第一块 IDE 硬盘称为 `/dev/hda`，第二块称为 `/dev/hdb`，等等。

硬盘还被划分为一个或多个分区，分区可以作为单独的设备。第一块硬盘的第一个分区称为/dev/hda1。如果有第二个分区，则称为/dev/hda2。SCSI 和 SATA 硬盘驱动器拥有自己的名称。第一块 SCSI 或 SATA 硬盘驱动器是/dev/sda，第二块是/dev/sdb，等等。同样，分区也进行编号，因此，第一块 SCSI 或 SATA 硬盘的第一个分区将是/dev/sda1。

SCSI 表示法有时候也用于其他类型的设备。一个常见的例子就是 USB 闪存，它被视为一个可移除的 SCSI 磁盘。基于这一原因，闪存的特殊文件也被命名为/dev/sda1 或相似的名称。

23.6 终端特殊文件：tty

在图 23-2 中，我们注意到有多种不同的特殊文件表示各种终端。下面说明具体原因。以前，终端是连接到主机计算机的独立物理设备(参见第 3 章)。这样的终端由名为/dev/tty1、/dev/tty2 等的特殊文件表示(正如第 7 章中解释的，缩写 TTY 是终端的同义词。这是因为第一种 Unix 终端就是 Teletype 机器，该机器被称为 TTY)。

现在，充当硬件设备的终端仍然使用/dev/tty 命名约定。特别地，当以单用户模式运行 Unix 时就是这种情况。键盘和显示器(控制台)充当内置的基于文本的终端。表示这个终端的特殊文件是/dev/tty1。同样，当在桌面环境中使用虚拟控制台(参见第 6 章)时，它也充当一个实际终端。默认情况下，Linux 支持 6 个这样的控制台，分别由特殊文件/dev/tty1 至 /dev/tty6 表示。

但是，当在窗口中使用 GUI 运行终端仿真程序时，事情又有所不同。因为没有实际终端，所以 Unix 创建所谓的伪终端(pseudo terminal, PTY)来模拟终端。当在 GUI 中打开终端窗口(参见第 6 章)，以及连接远程 Unix 主机(参见第 3 章)时，使用的就是 PTY。在两种情况中，PTY 都充当终端。

由于有两种不同的系统来创建伪终端，因此可以看到两种类型的名称。如果 Unix 使用的是第一种系统，那么表示伪终端的特殊文件将命名为类似于/dev/typ0 和/dev/typ1 的名称。如果使用的是另一种系统，那么这些特殊文件将命名为类似于/dev/pts/1 和/dev/pts/2 等的名称。图 23-2 中示范了两种类型的名称。

在任何时候，都可以使用 **tty** 程序显示终端的名称。例如，假设您正在使用虚拟终端 #3 进行工作。您输入了：

```
tty
```

输出为：

```
/dev/tty3
```

为了方便起见，特殊文件/dev/tty 表示当前正在使用的终端。例如，如果您正在使用虚拟控制台#3，那么/dev/tty 与/dev/tty3 相同。

下面举例说明如何使用特殊文件。从第 25 章中可知，使用 **cp** 程序可以复制文件。在第 11 章中，我们讨论了口令文件/etc/passwd。假设您希望复制一份口令文件，并将副本文

件命名为 **myfile**。完成该任务的命令为：

```
cp /etc/passwd myfile
```

这条命令就可以完成工作。下面考虑下述命令：

```
cp /etc/passwd /dev/tty
```

这将把口令文件复制到终端。结果是在显示器上显示口令文件的内容。试试看——然后想一想为什么发生这种情况。

23.7 伪设备特殊文件

我们将要讨论的最后一种类型的特殊文件是伪设备。伪设备是一个充当输入源或输出目标的文件，但是并不对应于实际设备(真实的或仿真的)。两个最有用的伪设备是 **null 文件**和 **zero 文件**。**null 文件**是 **/dev/null**；**zero 文件**是 **/dev/zero**。写入到这两种设备上的任何输出都会被抛弃。基于这一原因，有时候搞怪地称这些文件为“位桶(bit bucket)”

我们已经在第 15 章中详细讨论了 **null 文件**。下面是那一章中的一个例子。假设您有一个程序 **update**，该程序读取并修改大量的数据文件。在完成工作的过程中，**update** 显示所发生事情的统计信息。如果不希望看到统计信息，则可以将标准输出重定向到两个位桶中的一个上：

```
update > /dev/null
update > /dev/zero
```

如果希望体验一下，下面举一个可以立即体验的简单例子。输入命令：

```
cat /etc/passwd
```

这样可以看到口令文件的内容。现在，将 **cat** 命令的输出重定向到 **null 文件**或 **zero 文件**。注意，输出消失了：

```
cat /etc/passwd > /dev/null
cat /etc/passwd > /dev/zero
```

当处理输出时，两个位桶的作用相同。二者之间唯一的区别发生在输入上。当程序从 **/dev/null** 中读取数据时，不管请求输入多少字节，结果总是 **eof** 信号(参见第 7 章)。换句话说，读取 **/dev/null** 不返回任何东西。

当程序从 **/dev/zero** 中读取数据时，文件生成和请求一样多的字符。但是，它们都是值 **0**(数字 0)。在 Unix 中，认为这个值是 **null 字符**，或者更简单地说，是 **null**。尽管看上去非常奇怪，但是一个 **null 字符** 恒定源非常有用。例如，出于安全考虑，经常需要清除文件或整个磁盘的内容。在这些情况下，可以使用 **null 字符** 覆盖已有的数据，即从 **/dev/zero** 中向输出目标复制足够多的字节。

(这两个术语有点混乱：**null 文件**不返回任何东西，而 **zero 文件**返回 **null**。)

下面再举一个例子，在这个例子中，我们使用 **dd** 创建一个全新的文件，该文件完全由 **null** 字符构成(**dd** 程序是一个功能强大的 I/O 工具，这里不详细介绍这个工具。如果您想了解更多的信息，请查看联机手册)。

```
dd if=/dev/zero of=temp bs=100 count=1
```

在这个例子中，**if** 是输入文件，**of** 是输出文件，**bs** 是块大小，**count** 是块的数量。因此，我们从 **/dev/zero** 向文件 **temp** 中复制了 100 字节的数据块。

如果希望体验一下这个例子，可以运行 **dd** 命令，然后使用 **hexdump** 或 **od** 程序(第 21 章)显示 **temp** 的内容。当体验结束后，可以使用 **rm** 程序(第 25 章)移除 **temp** 文件。

最后两种伪设备是 **/dev/random** 和 **/dev/urandom**，这两个伪设备用来生成随机数。因此，每当需要随机数时，所需做的全部就是读取这两个文件之一。

或许您是那种在自己的个人生活中不太使用随机数的奇人。如果是这样的话，那么您可能奇怪它们为什么如此重要。答案就是，数学家和科学家使用这样的数字来创建存在概率的自然过程的模型。当处理概率问题时，一个不受限制、易于使用的随机数源是无价的(如果您对这类事情感兴趣，可以阅读一点有关“随机过程”的书籍)。另外，程序还使用随机数生成加密数据的加密键。

技术提示

Unix 和 Linux 提供两种不同的特殊文件来生成随机数：**/dev/random** 和 **/dev/urandom**。它们之间的区别比较微小，但是非常重要，特别是当完全的随机性至关重要时。

计算机化的随机数生成器收集“环境噪音(environmental noise)”，并将其存储在一个“熵池(entropy pool)”中。然后，使用熵池中的数据位生成随机数。如果熵池被耗尽，那么 **/dev/random** 文件将停止，等待收集更多的噪音。这样就可以确保关键操作(例如创建加密键)的完全随机性。但是，有时候，为了等待填充熵池，可能会存在延迟情况。

另一方面，**/dev/urandom** 文件永远不会停止生成随机数，即便熵池短缺(**u** 代表 unlimited，即无限)。为此，该文件会重用一些旧数据位。理论上，使用短缺熵池随机数加密的数据更易受到攻击。在实际中，区别并不是太大，因为没有人真的知道如何利用这样一个微小的理论缺陷*。当然，如果您是妄想狂，则可以使用 **random**，而不是 **urandom**，从而帮助您安心地睡觉。如果您不是妄想狂，则可以使用 **urandom**，其效果一样好，而且还不用等待。

23.8 命名管道：mkfifo

命名管道是一种用来创建特殊类型的管道设施的伪文件。通过这种方式，命名管道可以作为第 15 章所讨论的常规管道设施的扩展。在示范命名管道的工作机制之前，我们先快速地了解一下常规管道。下述命令提取系统口令文件中所有包含字符串“**bash**”的行。然后数据被管道传送给 **wc** 程序(参见第 18 章)，以统计这些行的数量：

* 至少在未分类的文献中。

```
grep bash /etc/passwd | wc -l
```

当以这种方式使用管道时，管道并没有具体的名称。管道将自动创建，且仅当两个进程正在运行时它才存在。基于这一原因，我们称之为**匿名管道**(anonymous pipe)。

命名管道与匿名管道相似，它们都将一个进程的输出连接到另一个进程的输入。但是，命名管道和匿名管道之间有两个重要的区别。首先，必须显式地创建命名管道。其次，当两个进程结束时，命名管道并不会消失。除非删除命名管道，否则它会一直存在。因此，一旦创建了命名管道，就可以反复地使用它。

通常将命名管道称为 FIFO(发音为“fie-foe”)，它是“先进先出，first-in first-out”的缩写。这是描述一种数据结构的计算机科学术语，在这种数据结构中，数据按进来的顺序被检索。更正式地讲，这样的数据结构称为**队列**(queue)。^{*}

在创建命名管道时，需要使用 **mkfifo**(make fifo)程序。该程序的语法为：

```
mkfifo [-m mode] pipe
```

其中 **mode** 是 **chmod** 程序使用的一种文件模式类型，**pipe** 是希望创建的管道的名称。(我们将在第 25 章讨论 **chmod** 时讨论文件模式，现在可以忽略 **-m** 选项)。

大多数时候，程序员使用命名管道来促进两个进程之间数据的交换，这种操作称为**进程间通信**(interprocess communication, IPC)。在这些情况中，程序将在需要时创建、使用然后删除命名管道。通过使用 **mkfifo** 程序，可以从命令行上手动创建命名管道。这样介绍有点泛泛而谈，所以下面示范一个例子，大家可以自己体验一下，领会它的工作方式。

为了进行体验，需要打开两个终端窗口或使用两个虚拟控制台。然后，在第一个终端窗口键入下述命令。该命令使用 **mkfifo** 创建命名管道 **fifotest**：

```
mkfifo fifotest
```

下面，我们向该管道发送一些输入。在同一个窗口中，输入下述命令，在系统口令文件中 **grep** 包含“bash”的行，并将输出重定向到 **fifotest**：

```
grep bash /etc/passwd > fifotest
```

现在转到第二个终端窗口或虚拟控制台。输入下述命令，从命名管道中读取数据，并统计行的数量。输入了该命令以后，**wc** 程序就会从命名管道中读取数据并显示它的输出。

```
wc -l < fifotest
```

一旦使用完命名管道，就可以删除它。为了删除命名管道，可以使用 **rm**(remove, 移除)程序：

```
rm fifotest
```

很明显，这是一个人为的例子。毕竟，我们在不久前使用一行简单的命令就完成了同一件事情。

^{*} 这是相对于栈(参见第 8 章和第 24 章)而言的，在栈中，元素按 LIFO(后进先出，last in first out)的方式存储和检索。

```
cat /etc/passwd | wc -l
```

但是，您至少理解了命名管道的工作方式。考虑一下，命名管道对要求进程间通信的程序员来说会有多大的价值。程序员需要做的就是让程序创建一个命名管道，然后在需要时用它来从一个进程向另一个进程传送数据。一旦工作完成，程序就可以移除管道。这种方法简单、容易并且可靠，同时不需要创建中间文件来存储过渡数据。

23.9 proc 文件

proc 文件是那些提供一种简单的途径来检查多种类型的系统信息的伪文件，proc 文件直接从内核获取信息，而不是使用复杂的程序搜出数据。原始的 proc 文件系统是为了提取进程*的信息而开发的，因此命名为“proc”。

所有的 proc 文件都存放在 **/proc** 目录中。在这个目录中，可以发现系统中每个进程对应一个子目录。子目录的名称非常简单，就是各种进程的进程 ID(正如第 26 章中将讨论的，每个进程都有一个唯一的标识号，称为“进程 ID”)。例如，假设现在您的系统上的一个进程是#1952。那么有关这个进程的信息可以在 **/proc/1952** 目录下的伪文件中找到。

/proc 目录的思想取自 Plan 9 操作系统**，一个从 20 世纪 80 年代中期到 2002 年的研究项目。Plan 9 项目由创建 Unix、C 和 C++的同一组研究人员在贝尔实验室建立。Plan 9 中的基本概念之一就是所有的系统界面都是文件系统的一部分。其中，有关进程的信息位于 **/proc** 目录中。尽管 Plan 9 并没有取得成功，但是以文件访问许多类型的信息的思想非常引人注目，因此，被 Unix 和 Linux 社区广泛采纳。

事实上，Linux 不仅采纳了 **/proc**，而且还极大地扩展了它。现代的 Linux 系统使用该目录来存放许多其他伪文件，提供众多内核数据的访问功能。实际上，如果是超级用户，甚至还有可能通过写 proc 文件改变一些 Linux 内核的值(千万不要这样做)。图 22-3 示范了 Linux 使用的最有趣的 proc 文件。可以使用与显示普通文本文件内容相同的方式来显示这些文件中的信息。例如，为了显示有关处理器的信息，可以使用下述命令之一：

```
cat /proc/cpuinfo
less /proc/cpuinfo
```

* 正如第 6 章所述，进程的思想是 Unix 的基本思想。实际上，在 Unix 系统中，每个对象都或者由文件表示，或者由进程表示。用简单的术语讲，文件存储数据，或者允许访问资源；而进程是正在执行的程序。

进程更精确的定义(同样参见第 6 章)就是一个装载到内存中并且准备运行的程序，随同程序一起装载的是跟踪程序所需的数据和信息。

** 名称 Plan 9 来源于一个严重拼凑的科幻电影 *Plan 9 from Outer Space*，人们普遍认为这是一部最糟糕的电影。为什么这样一群技能高度丰富、富有想象力的计算机科学家以这样一部电影来命名一个重大的项目呢？我能说的就是这一定是在开玩笑。

proc 文件	相关信息
<code>/proc/xxx/</code>	进程#xxx
<code>/proc/cmdline</code>	内核选项
<code>/proc/cpuinfo</code>	处理器
<code>/proc/devices</code>	设备
<code>/proc/diskstats</code>	逻辑磁盘设备
<code>/proc/filesystems</code>	文件系统
<code>/proc/meminfo</code>	内存管理
<code>/proc/modules</code>	内核模块
<code>/proc/mounts</code>	已挂载设备, 挂载点
<code>/proc/partitions</code>	磁盘分区
<code>/proc/scsi</code>	SCSI 和 RAID 设备
<code>/proc/swaps</code>	交换分区
<code>/proc/uptime</code>	内核运行时间(秒), 内核空闲时间(秒)
<code>/proc/version</code>	内核版本、分发、gcc 编辑器(用来构建内核)

图 23-3 Linux 中最有趣的 proc 文件

proc 文件是用来访问内核信息的伪文件。这样的文件主要由系统程序使用。尽管极少直接使用 proc 文件, 但是有几个 proc 文件非常有趣, 需要知道它们。详情请参见正文。

通常, 除非是系统管理员, 否则永远不需要查看 proc 文件。在大多数情况下, proc 文件只由那些需要内核的高度技术化信息的程序使用。例如, 在第 26 章中, 我们将讨论 **ps**(process status, 进程状态)程序, 该程序显示系统上进程的信息。**ps** 程序正是通过读取合适的 proc 文件, 收集所需数据的。

另外还有一个特别引人注目的 proc 文件, 这个文件没有在图 23-3 中列出, 它就是 `/proc/kcore`。这个文件表示计算机的实际物理内存。可以使用 **ls** 程序加 **-l** 选项(参见第 24 章)显示这个文件的大小:

```
ls -l /proc/kcore
```

这个文件看上去非常巨大, 实际上, 它同计算机上的整个内存(RAM)的大小相同。但是, 记住, 这是一个伪文件: 它实际上不需要占用空间。

提示

Linux 用户: 我鼓励您研究自己系统上的 proc 文件。通过查看这些文件, 可以自学许多系统的配置情况以及工作方式。为了避免产生问题, 一定要保证不以超级用户登录, 即便只是为了查看一下。

23.10 树型结构文件系统：文件系统层次结构标准

在接下来的几节中，将讨论 Unix 的文件系统以及它的组织方式。在我们的讨论中，将使用标准的 Linux 文件系统作为示例。各种 Unix 系统之间的细节可能有所不同，因此您的系统可能与这里介绍的内容有点不同。但是，基本思想，包括大多数目录名称都是相同的。

典型的 Unix 包含大约 100000 个文件*，这些文件存储在各个目录和子目录中。所有这些文件被组织成文件系统，在文件系统中，目录基于一个称为“根目录”的主目录组织成树形结构。文件系统的任务就是存储和组织数据，并向用户和程序提供数据的访问功能。图 23-4 示范了一个文件系统组织图。在该图中，根目录是系统中所有其他目录的父目录直接或间接。

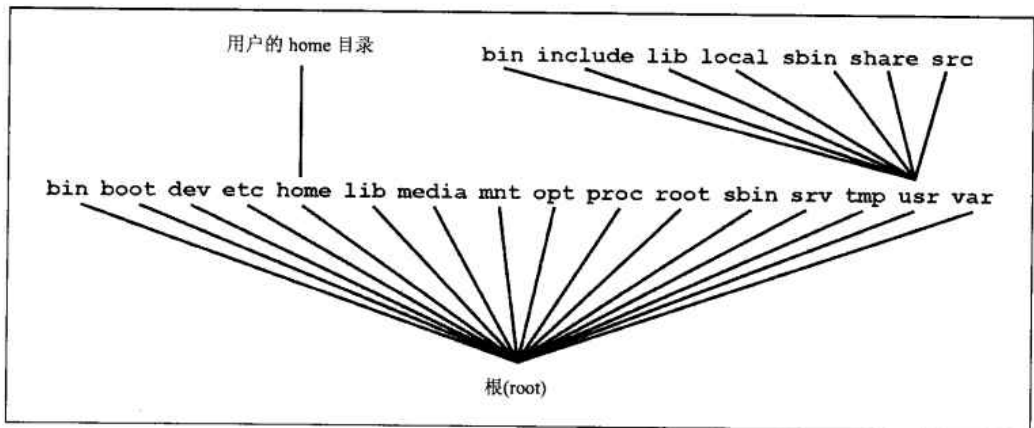


图 23-4 标准 Linux 文件系统

Unix 和 Linux 系统中包含大量的文件——至少有 100000 到 200000 个文件，这些文件组织成一个目录和子目录的树形结构集合。本例只示范了标准 Linux 文件系统的框架。

这里示范的例子遵循文件系统层次结构标准(本章后面描述)。可以看出，根(主)目录包含 16 个子目录，而每个子目录又包含有自己的子目录，这些子目录中又包含有子目录，等等。本图还列出了 `/usr` 目录中通常存在的 7 个子目录。

第一次看到 Unix 文件系统组织结构时，可能会有点心虚。毕竟，Unix 文件系统组织结构中的名称都比较奇怪和神秘，而且没有什么直观含义。但是，像 Unix 其他大部分领域一样，一旦理解了 Unix 文件系统的模式和工作方式，那么 Unix 文件系统就会非常易于理解。在本章后面，我们将介绍图 23-4 中所示的每个子目录，每次一个。同时，我们还会解释如何使用它们，以及它们名称的含义。

但是，在开始之前，我希望花点时间解释一下 Unix 文件系统是怎样采用这种方式来

* 我并没有进行夸大。一些基本的 Unix 系统提供的文件超过 200000 个。为了估计系统中文件和目录的数量，可以以超级用户的身份运行下述命令：

```
ls -R / | wc -l
```

组织的。正如第 2 章中讨论的，第一个 Unix 系统于 20 世纪 70 年代初在贝尔实验室开发。图 23-5 示范了原始 Unix 系统的结构，这个系统一开始就设计为分层次的树形结构。现在还不用关心文件名称的含义，稍后会解释它们。我希望大家注意的是原始的文件系统看上去就像标准文件系统(图 23-4)的一个子集。

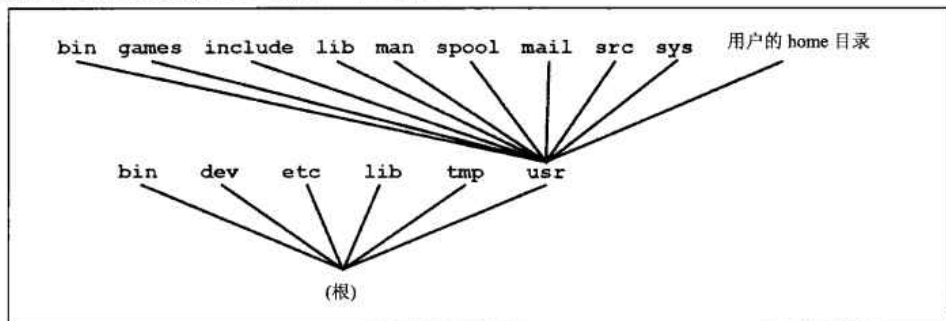


图 23-5 原始 Unix 文件系统

从一开始，Unix 文件系统就使用目录和子目录组织文件。这张图示范了 20 世纪 70 年代初原始 Unix 系统使用的树形结构。注意该树或多或少地像当前文件系统(图 23-4)的一个子集。

随着 Unix 系统的发展，Unix 文件系统的组织也逐渐因为需求的变化以及各种 Unix 开发人员的爱好而开始改变。尽管基本的格式仍然相同，但是各种版本的 Unix 之间在细节上已经不同。这就产生了不少混乱，特别当用户交替使用 System V Unix 和 BSD(参见第 2 章)时。20 世纪 90 年代，当各种 Linux 发行版的创建者开始推出他们自己的变种时，情况更加混乱。

在 1993 年 8 月，一群 Linux 用户组建了一个小组织，准备开发一个标准的 Linux 目录结构。他们在 1994 年 2 月发布了第一个这样的标准。在 1995 年初，BSD 社区的成员加入他们的小组，因此他们扩展了自己的目标。从那时起，他们致力于为所有的 Unix 系统创建一个标准的文件系统组织，而不再仅限于 Linux。新的系统称为文件系统层次结构标准 (Filesystem Hierarchy Standard, FHS)。当然，因为没有 Unix 警察，所以这个标准是自愿的。不过，许多 Unix 和 Linux 开发人员选择采纳了 FHS 的大部分内容。

尽管许多 Unix 系统在一些方面与 FHS 不同，但是 FHS 是一个精心设计的规划，而且它确实抓住了现代 Unix 和 Linux 文件系统组织的本质。如果理解了 FHS，使用所遇到的任何类型的 Unix 文件系统都会变得非常简单。基于这一原因，在讨论 Unix 文件系统的细节时，我们将使用 FHS 作为模型。如果希望看一看基本的 FHS 是什么样子，则可以参考图 23-4。

提示

从图 23-4 可以看出，除了根目录之外，所有的目录都位于另一个目录之内。因此，从技术上讲，除了根目录之外，所有的目录都是子目录。

但是，在日常生活中，通常只说目录。例如，我们可以谈论“/bin 目录”。只有当希望强调特定的目录位于另一个目录之中时，才使用术语“子目录”，例如，“/bin 目录是根目录的一个子目录。”

23.11 根目录；子目录

从一开始，Unix 文件系统就组织成一个树形结构。在第 9 章中，我们将树作为抽象数据结构进行了讨论，那时，我提到了树的主节点称为根(详情请参见第 9 章)。基于这一原因，我们称 Unix 文件系统的主目录为根目录。

因为根目录非常重要，所以它的名称经常成为命令的一部分。但是，总是键入字符串“root”肯定会很烦人，所以我们使用一个/(斜线)来表示根目录。下面举一个简单的例子来示范它的工作方式。为了列举特定目录下的文件，我们可以使用 **ls** 程序(参见第 24 章)。只需键入 **ls**，后面跟着目录的名称即可。列举根目录中所有文件的命令为：

```
ls /
```

当指定根目录中的目录或者文件的名称时，需要先写一个/，后面跟名称。例如，在根目录中，有一个子目录 **bin**。为了列举这个目录中的全部文件，可以使用命令：

```
ls /bin
```

正式地讲，这意味着“位于/(根)目录之中的目录 **bin**”。

为了表示一个目录或者文件位于另一个目录之中，需要使用一个/将名称分隔开。例如，在 **/bin** 目录中，有一个包含 **ls** 程序本身的文件。这个文件的正式名称是 **/bin/ls**。同样，在 **/etc** 目录中，可以发现 Unix 口令文件 **passwd**(参见第 11 章)。这个文件的正式名称是 **/etc/passwd**。

当谈论这样的名称时，我们将/字符发音为“slash，斜线”。因此，名称 **/bin/ls** 的发音为“slash-bin-slash-L-S”。

提示

在适应这一命名法之前，使用/字符可能会产生混乱。这是因为/拥有两层含义，这两层含义各不相关。

在文件名的开头，/代表根目录。在文件名的中间，/充当定界符(好好地想一想)。

名称含义

root

在第 4 章中，我们解释过为了成为超级用户，需要使用用户标识 **root** 登录。现在您应该明白这个名称的来源了：超级用户的用户标识根据根目录命名，而根目录是文件系统中最重要的目录。

23.12 挂载文件系统：mount、umount

在 Unix 文件系统中，数十万个文件组织成一棵非常大的树，树的基就是根目录。在大多数情况下，并不是所有的文件都存储在同一个物理设备上。更准确地说，文件存储在许多不同类型的设备上，包括多个磁盘分区(正如前面所解释的，每个磁盘分区被当作一个单独的设备)。

每个存储设备都有自己的本地文件系统，其目录和子目录按照标准 Unix 方式组织成树。但是，在访问本地文件系统时，它的树必须附加到主树上。这可以通过将小型文件系统的根目录连接到主文件系统中的特定目录上来实现。当采用这种方式连接小型文件系统时，我们称**挂载(mount)**该文件系统。小文件系统在主树中附加到的目录称为**挂载点(mount point)**。最后，当断开文件系统时，称之为**卸载(umount)**文件系统。

作为启动过程的一部分，Unix 系统每次启动时，都会自动挂载一些本地文件系统。因此，当系统启动完毕以后，主文件系统已经添加了几个其他文件系统。

有时候，可能不得不手动挂载一个设备。这样做时，需要使用 **mount** 程序。而如果要卸载设备，则需要使用 **umount**。作为一个普通的预防措施，只有超级用户才允许挂载文件系统。但是，出于方便考虑，一些系统被配置为允许普通用户挂载特定的预设置设备，例如 CD 或 DVD。

下面举一个 **mount** 命令的例子。在这个例子中，我们挂载设备 **/dev/fd0** 中的软盘驱动器文件系统，并将其附加在主树中的 **/media/floppy** 位置上：

```
mount /dev/fd0 /media/floppy
```

该命令的结果就是允许用户通过 **/media/floppy** 目录访问软盘上的文件。

挂载和卸载是系统管理任务，要求用户是超级用户，因此这里不再详细解释。如果想了解这方面的知识，可以参阅联机手册(**man mount**)。但是，作为一名普通用户，可以显示当前挂载到系统上的所有文件系统列表，为此，输入 **mount** 命令本身即可：

```
mount
```

广义地讲，存储设备有两种类型。固定介质(**fixed media**)永久附属于计算机，例如硬盘驱动器。**可移动介质(removable media)**在系统运行时可以改变，例如 CD、DVD、软盘、磁带、闪存、存储卡等。在系统层次，该区别非常重要，因为文件系统存在消失的可能性，Unix 系统必须确保恰当地管理该文件系统。例如，在允许弹出 CD 之前，Unix 必须确保所有挂起的输出操作都已经完成。

基于这一原因，文件系统层次结构标准要求使用特定的目录挂载文件系统。对于没有挂载在其他位置的固定介质(例如额外的硬盘)来说，指定目录是 **/mnt**；对于可移动介质来说，目录是 **/media**。

名称含义

挂载、卸载

在 Unix 初期(大约 1970 年)，磁盘驱动器是大型、昂贵的设备(按现在的标准来说)，而且存放相对来说少量的数据(最多 40 兆字节)。与现代的磁盘驱动器不同——现代的磁盘驱动器是一个整体，以前的硬盘驱动器使用可移动的“盘组(disk pack)”，每个盘组都有自己的文件系统。

每当用户需要改变盘组时，系统管理员不得不动手在物理上卸载当前盘组，挂载上新的盘组。这就是为什么时至今日，我们还谈论“挂载”和“卸载”一个文件系统的原因。当使用 **mount** 程序时，我们执行的是与在驱动器中挂载一个盘组等价的软件操作。

23.13 漫游根目录

理解计算机文件系统的最快捷方式就是查看根目录，并检查所有的子目录。这些目录形成整个系统的骨干。就这一点而论，有时候也称它们为顶级目录(toplevel directory)。

图 23-6 汇总了本章前面讨论的文件系统层次结构标准(FHS)所指定的根目录的标准内容。尽管各个 Unix 系统中这一内容可能有所不同，但是可以认为所有现代的文件系统都是 FHS 的变体。因此，一旦理解了 FHS，就能够理解遇到的任何 Unix 文件系统。

目录	内容
/	根目录
/bin	基本程序
/boot	启动系统时所需的文件
/dev	设备文件
/etc	配置文件
/home	用户的 home 目录
/lib	基本共享库，内核模块
/lost+found	由 fsck 恢复的受损文件
/media	可移动介质的挂载点
/mnt	不能挂载在其他位置上的固定介质的挂载点
/opt	第三方应用程序(“可选软件”)
/proc	proc 文件
/root	根用户(超级用户)的 home 目录
/sbin	由超级用户运行的基本系统管理程序
/srv	本地系统所提供服务的数据库
/tmp	临时文件
/usr	静态数据使用的辅助文件系统
/var	可变数据使用的辅助文件系统

图 23-6 根目录的内容

Unix 文件系统的框架由顶级目录，也就是根目录的子目录所创建。本图中所列举的目录是文件系统层次结构标准(FHS)要求的全部顶级目录。因为一些 Unix 和 Linux 系统没有完全遵循 FHS，所以您看到的可能与这里示范的有所不同。然而，您可以将这个列表作为理解自己系统的起点。

这里我们的目标就是从根目录开始，遍历所有的顶级目录。在讨论每个目录的过程中，都可以使用 **ls** 程序(参见第 24 章)在自己的系统上查看一下这个目录。例如，为了显示 **/bin** 目录的内容，可以使用下述两条命令中的一条：

```
ls /bin
ls -l /bin
```


当不加选项地使用 **ls** 时, 只能看到文件名。如果使用 **-l(long, 长)** 选项, 则可以看到额外的信息。如果输出太多, 输出信息将快速地滚过屏幕, 所以可以将输出管道传送给 **less** 程序(参见第 21 章)每次一屏地显示:

```
ls -l / | less
ls -l /bin | less
```

根目录: 根目录是整个文件系统的基础。在许多 Unix 系统中, 根目录只包含子目录。在一些系统上, 根目录还包含一个普通文件: 内核(参见后面对 **/boot** 的讨论)。

/bin: 这个目录存放最重要的系统程序, 即系统管理员在单用户模式(参见第 6 章)下管理系统所需的基本工具。这些工具都是可执行文件, 正如本章前面所讨论的, 可执行文件也是二进制文件。因此该目录命名为 **bin**, 即存放二进制(binary)文件的位置。更简单地讲, 可以认为该目录是程序的存储箱。该目录中的一些程序也可以由常规用户使用。

/boot: 这是系统存放引导过程(参见第 2 章)中所需全部文件的位置。内核必须位于这个目录或者根目录中。内核非常容易找到: 输入下述命令, 查找拥有一个奇怪名称的大文件即可。

```
ls -l /boot | less
ls -l /
```

如果升级过系统, 则会在该目录中发现不止一个版本的内核。在大多数情况下, 正在使用的内核是最新的内核, 通过查看其名称可以进行确认(内核的版本号是名称的一部分)。

/dev: 所有的特殊文件都位于这个目录中。大多数特殊文件都代表物理设备; 少数一些特殊文件代表伪设备(参见本章前面的讨论)。这个目录中还包含一个叫 **makedev** 的程序, 该程序用来创建新的特殊文件。

/etc: 这个目录中包含的是配置文件。正如第 6 章中讨论的, 配置文件是某程序启动时处理的文本文件, 其中包含有影响程序操作的命令或信息。配置文件在本质上与第 14 章中讨论的 **rc** 文件相似。

/home: 当创建 Unix 账户时(参见第 4 章), 管理员会为用户标识赋予一个“home 目录”。home 目录是存放个人文件和目录的位置。home 目录的名称与用户标识相同。因此, 如果用户标识是 **harley**, 那么 home 目录就是 **/home/harley**。我们将在本章后面详细地讨论 home 目录。所有的 home 目录都位于 **/home** 中, 但是有一个例外。这个例外就是超级用户的 home 目录, 它的 home 目录位于 **/root** 中(参见下面)。

/lib: 当程序运行时, 经常要调用库(library), 即已经存在的数据和代码模块。Unix 提供有大量的库, 以允许程序访问操作系统提供的服务。这个目录中包含运行 **/bin** 和 **/sbin** 目录中的程序所需的基本库和内核模块。

/lost+found: 如果 Unix 没有正常地关机, 那么那些仅完成部分写入的文件将受到损坏。Unix 下一次启动时, 一个叫 **fsck(filesystem check, 文件系统检查)** 的程序将自动运行, 检查文件系统并修复问题。如果发现损坏的文件, 那么 **fsck** 将挽救这些文件, 并将它们移动到 **/lost+found** 目录中。然后, 系统管理员可以查看恢复的文件, 适当地处置它们。

/media: 这里是可移动介质(例如 CD、DVD、软盘、闪存、存储卡等)的挂载点(参见

本章前面有关挂载文件系统的讨论)。

/mnt: 这是不会在其他位置挂载的固定介质(例如额外的硬盘)的挂载点。曾经, 这里是唯一的挂载目录, 因此经常可以看到有人在这里挂载可移动介质。除非您希望向他人显示您什么也不知道, 否则不要模仿这样的人, 因为可移动介质属于 **/media** 目录。

/opt: 这个目录是安装第三方应用程序的位置(名称 **/opt** 代表 “optional software, 可选软件”)。在 **/opt** 中, 每个程序都根据自己的需要拥有自己的子目录。这就给予应用程序开发人员一个指定的位置来安装他们的软件, 同时不必担心特定文件系统的组织方式。**/opt** 中的子目录或者根据公司命名, 或者根据应用程序命名。为了使事情有序, LANANA(Linux Assigned Names and Numbers Authority, 参见本章前面的讨论)维护了一个官方的名称列表。如果希望查看该列表, 可以在网络上搜索 “LSB Provider Names” 或 “LSB Package Names” (LSB 代表 “Linux Standard Base”)。

/root: 这是超级用户, 也就是用户标识 **root** 的 home 目录。其他所有的 home 目录都位于 **/home** 目录中(参见上面)。

/sbin: 名称 **/sbin** 代表 “system binaries, 系统二进制文件”。这个目录中存放用于系统管理的程序。通常, 这个目录中的程序必须由超级用户运行。

/srv: 这个目录是为与本地提供的服务(service, 因此命名为 **/srv**)相关的数据保留的。在这里存储数据的典型服务包括 **cgi**、**Web**、**ftp**、**cvs**、**rsync** 等。

/tmp: 这个目录用于临时存储。任何人都可以在这个目录中存储文件。但是, 最终, **/tmp** 的内容将自动移除。基于这一原因, 程序通常仅使用这个目录来存放只需短时间保存的文件。

/usr: 这个目录是辅助文件系统的根, 包含有辅助文件系统的重要子目录。**/usr** 的目的是用来存放静态数据, 即没有系统管理员的干涉不会改变的数据。根据其特性, 静态数据不会随时间改变。这就允许 **/usr** 驻留于自己的设备, 该设备甚至有可能是诸如 **CD-ROM** 之类的只读设备。以前, **/usr** 是存放用户 home 目录的目录。现在 **/usr** 只用来存放静态数据, home 目录需要放在 **/home** 中(参见上面)。

/var: 这个目录的名称意味着 “variable, 可变”。像 **/usr** 一样, 这个目录是辅助文件系统的根, 包含有辅助文件系统的重要子目录。两者之间的区别在于 **/usr** 存放的是静态数据, 而 **/var** 存放的是可变数据, 即随着时间改变的数据, 包括日志文件(记录系统发生的事情的文件)、打印文件、电子邮件消息等。像 **/usr** 一样, **/var** 文件系统通常驻留于自己的设备。通过这种方式将静态数据和可变数据分离开来, 可以使系统易于管理。例如, 系统管理员可以创建一个备份系统, 经常(相对于静态文件)单独保存可变文件。

名称含义

dev、**etc**、**lib**、**mnt**、**opt**、**src**、**srv**、**tmp**、**usr**、**var**

Unix 传统喜欢为文件系统的顶级目录使用 3 个字母的名称。原因就是这样的名称比较短, 易于键入。但是, 当谈论这些名称时, 往往不易于发音。基于这一原因, 每个 3 字母的名称都有一个首选的发音。

dev: “dev”

etc: “et-cetera” 或者 “et-see”

lib: “libe” (和 “vibe” 押韵)

mnt: “mount”

opt: “opt”

src: “source”

srv: “serv”

tmp: “temp”

usr: “user”

var: “var” (和 “jar” 押韵)

通常，如果在谈论与 Unix 相关的事情，当碰到缺 1 个或 2 个字母的名称时，应该在发音时加上缺的字母。例如，文件 `/etc/passwd` 的名称就发音为 “slash et-cetera slash password”。文件 `/usr/lib/X11` 就发音为 “slash user slash libe slash x-eleven”。

有时候，可能会听到有人说 **etc** 代表 “extended tool chest，扩展工具箱”，或者 **usr** 的含义是 “Unix system resources，Unix 系统资源” 等。这些故事都不是真实的。这些名称都是缩写，而不是只取首字母的缩写词。

23.14 漫游/usr 目录

正如前面讨论的，**/usr** 和 **/var** 目录都是集成到主文件系统中的单独文件系统的挂载点。**/usr** 文件系统针对的是静态数据，**/var** 针对的是可变数据。这两个目录都是用来存放系统数据的，与系统数据相对应的是用户数据，它们存放在 **/home** 目录中。

另外，这两个目录也包含大量的标准子目录。但是，**/var** 文件系统主要针对系统管理员，所以对普通用户来说没有那么重要。另一方面，**/usr** 文件系统更有趣，它包含有对常规用户和程序员来说都非常有用的文件。基于这一原因，我们将快速地浏览一遍 **/usr** 目录，了解文件系统层次结构标准中描述的最重要的子目录。出于参考目的，图 23-7 中汇总了这些目录。正如前面所说的，您自己的系统可能与标准布局有所不同。

目录	内容
/usr/bin	非基本程序(大多数用户程序)
/usr/include	C 程序的头文件
/usr/lib	非基本共享库
/usr/local	本地安装程序
/usr/sbin	由超级用户运行的非基本系统管理程序
/usr/share	共享系统数据
/usr/src	源代码(只用于参考)

图 23-7 /usr 目录的内容

/usr 目录是包含对用户和程序员都有用的静态数据的辅助文件系统的挂载点。这里列举的是按照文件系统层次结构标准，这个目录中最重要的子目录。详情请参见正文。

/usr/bin: 与根目录中的目录/bin 相同, 这个目录包含的也是可执行程序。这个目录要比/bin 包含更多的程序。实际上, /usr/bin 是系统中大多数可执行程序的存放位置。例如, 在我的一个 Linux 系统上, /bin 目录中只存放有 100 个程序, 而/usr/bin 目录中存放着 2084 个程序。

/usr/games/: 这是整个 Unix 文件系统中我最喜欢的目录(或许除/home/harley 之外)。顾名思义, 这个目录中包含的是游戏。该目录中还包含有大量的娱乐和教育程序。以前, /usr/games 目录中存放着各种有趣、令人愉快的程序。我最喜爱的游戏是 **adventure**, 最喜爱的娱乐是 **fortune**。如果在 Internet 上浏览一下, 仍然可以发现这些程序——以及更多其他程序, 可以下载这些程序并在自己的系统上安装它们(当在 Internet 上查找游戏时, 看看能不能找到“Hunt the Wumpus”)。现在大多数游戏已经不存在了, 这真是可怕的遗憾。许多 Unix 系统只提供少数几个游戏, 一些系统甚至不提供游戏。好像大多数人已经忘了 Unix 就是用来娱乐的。

/usr/include/: 这是 C 和 C++程序员所使用的包含文件(include file)的存储区。包含文件有时候称为头文件(header file), 包含有源代码, 程序员在需要时可以使用。典型的包含文件中拥有例程、数据结构、变量、常量等内容的定义。我们使用“包含文件”这一名称, 是因为在 C 语言中, 需要使用#include 语句将这样的文件集成到程序中。名称“头文件”来自于一个事实, 即这样的文件通常包含在程序的开头。包含文件的扩展名是.h, 例如 **ioctl.h** 和 **stdio.h**。

/usr/lib: 与目录/lib 相同, 这个目录中存放的也是库, 即已经存在的数据和代码模块, 程序使用它们访问操作系统提供的服务。

/usr/local: 这个目录是为系统管理员准备的, 系统管理员使用它来支持本地用户。这个目录是存放本地程序和文档资料的位置。这个目录的一个典型应用就是创建一个子目录 **/usr/local/bin**, 用来存放不属于主系统的程序。将软件存放在这里可以确保在系统升级时, 不会覆盖软件。

/usr/sbin: 同/sbin, 这个目录中包含有系统管理员使用的系统程序。从概念上讲, /usr/sbin 和/sbin 的关系就如同/usr/bin 和/bin 的关系一样(参见上面有关/usr/bin 的讨论)。

/usr/share: 有许多文件包含有静态数据——文档资料、字体、图标等, 它们需要在用户和程序之间共享。/usr/share 目录中含有大量的子目录用来存放这样的文件。例如, 在第 20 章中我们讨论了字典文件。在许多系统中, 可以在/usr/share/dict/words 中找到这个文件。最有趣的共享文件就是含有第 9 章中所讨论的联机文档资料的文件。其中, Unix 手册存储在/usr/share/man 中, Info 系统存储在/usr/share/info 中。

/usr/src: 名称 **src** 代表“source code, 源代码”。在这个目录中, 可以发现一些包含有系统源代码的子目录, 这些代码通常只是用来参考。在许多 Linux 系统上, 内核的源代码位于/usr/src/linux 中。

/usr/X11: 这个目录中存放着大量由 X Window(支持 GUI 的系统, 参见第 5 章)使用的文件和目录。在一些系统上, 这个目录名为/usr/X11R7 或者/usr/X11R6(如果系统比较老的话)。

23.15 使用多个目录存放程序的原因

正如前面所讨论的, Unix 中使用两个不同的目录来存放常用的可执行程序: **/bin** 和 **/usr/bin**。您可能会奇怪, 为什么 Unix 提供两个这样的目录? 为什么不把所有的程序都存储在一个目录中呢? 答案就是存放在两个 **bin** 目录中是由历史原因造成的。

在 20 世纪 70 年代初期, 前几版本的 Unix 是在贝尔实验室的 PDP 11/45 微型计算机(参见第 2 章)上开发的。Unix 开发人员使用的 PDP 11/45 拥有两个数据存储设备。主要设备是带固定头的磁盘, 通常称为磁鼓。磁鼓相对较快, 因为在磁盘旋转时读写头不用移动。但是, 数据存储量不会超过 3MB。

辅助设备是一种常规的磁盘, 称为 RP03。RP03 磁盘上的读写头在磁道之间来回移动, 从而允许存储更多的数据——高达 40MB。但是, 因为读写头的移动, 该磁盘的速度要比磁鼓慢一些。

为了在一台计算机上提供多个存储设备, Unix 开发人员采用了一种设计。在这种设计中, 每个设备都拥有自己的文件系统。主设备(磁鼓)中存放的文件系统称为根文件系统; 辅助设备(磁盘)上存放的文件系统称为 **usr** 文件系统。

理想情况下, 最好将整个 Unix 系统都存放在磁鼓上, 因为它要比磁盘快一些。但是, 磁鼓上没有足够的空间。所以, Unix 开发人员将所有的文件分成两组。第一组文件由启动进程及运行裸操作系统所需的文件构成。这些文件存储在磁鼓上的根文件系统中。其余文件存储在磁盘上的 **usr** 文件系统中。

在启动过程中, Unix 从磁鼓上引导。这样操作系统就可以立即访问根文件系统中的基本文件。一旦 Unix 启动完毕并运行, 它将挂载 **usr** 文件系统, 从而能够访问其他的文件。

这两个文件系统都拥有一个 **bin** 目录来存放可执行程序。根文件系统的是 **/bin**, **usr** 文件系统的是 **/usr/bin**。在启动过程中, 在挂载 **usr** 文件系统之前, Unix 只能访问根文件系统中相对较少的存储区。基于这一原因, 将基本程序存储在 **/bin** 中, 而其他程序存储在 **/usr/bin** 中。同样, 库文件也分成两个目录存储: **/lib** 和 **/usr/lib**。临时文件也分成两个目录存储: **/tmp** 和 **/usr/tmp**。在所有情形中, 根文件系统只存放最重要的文件, 即启动和解决问题所必须的文件。其他的文件都存储在 **usr** 文件系统中。

现在, 存储设备已经十分快速, 廉价而且还可以存储大量的数据。客观来说, 已经没有理由再将 Unix 的核心分成多个文件系统, 并存储在多个设备上了。实际上, 一些 Unix 系统将所有通用的二进制文件存储在一个大目录中。然而, 许多 Unix 系统依然使用单独的文件系统来组合成一棵大树。我们将在本章后面讨论虚拟文件系统时再讨论这样设计的原因。

通常, 现代的 Unix 操作系统将软件分为 3 种类型: 通用的程序, 可以由任何人使用; 系统管理程序, 只能由超级用户使用; 大型的第三方应用程序, 这些程序要求大量的文件和目录。正如本章前面所讨论的, 3 种不同类型的程序分别存储在各自的目录中。出于参考目的, 图 23-8 汇总了用来查找 Unix 程序文件的各种不同位置。

通用程序	
/bin	基本程序
/usr/bin	非基本程序
/usr/local/bin	本地安装程序
系统管理程序	
/sbin	由超级用户运行的基本系统管理程序
/usr/sbin	由超级用户运行的非基本系统管理程序
/usr/local/sbin	本地安装的系统程序
第三方应用程序	
/opt/xxx	应用程序 xxx 的静态数据, 包括程序
/var/opt/xxx	应用程序 xxx 的可变数据

图 23-8 存放程序文件的目录

Unix 文件系统拥有许多存放程序文件的不同位置。通用程序存储在名为 **bin**(binary files, 二进制文件) 的目录中。系统管理程序存储在名为 **sbin**(system binaries, 系统二进制) 的目录中。大型的第三方应用程序存储在名为 **opt**(optional software, 可选软件) 的目录中。程序再根据是基本程序还是非基本程序进一步细分。基本程序是启动系统或者执行关键系统管理所必需的程序。其他程序是非基本程序。

这里所示范的细节都基于文件系统层次结构标准。您自己的系统可能与此存在差别。

23.16 home 目录

除了这么多装满重要文件的系统目录之外, 很显然也需要给用户提供一个有序的系统, 用于控制个人文件的存储。当然, 对于像您我这样聪明的人来说, 即便允许将自己的私人程序存放在 **/bin** 目录, 或者将自己的私人数据存放在 **/etc** 目录中也不会把事情搞糟。但是, 对于大多数人, 我们不允许他们随意地将文件想放在哪就放到哪, 我们需要更有组织性。

解决方法就是给每个用户一个 **home** 目录, 一个可以任由用户自己处理的目录。当创建 Unix 账户(参见第 4 章)时, 系统就会自动地创建一个 **home** 目录。**home** 目录的名称存放在口令文件(参见第 11 章)中, 当登录时, 系统会自动地将您置于这个目录之中(在阅读了第 24 章之后, 您就会理解位于目录之“中”这一思想)。在自己的 **home** 目录中, 可以根据自己的需要存储文件以及创建其他子目录。实际上, 许多人都拥有自己精心设计的树型结构, 而所有这些都位于自己的 **home** 目录中。

文件系统层次结构标准建议 **home** 目录最好在 **/home** 目录中创建。在小型系统上, **home** 目录的名称就是用户标识的名称, 例如 **/home/harley**、**/home/linda** 等。在有許多用户标识的大型系统上, 可能还有额外的子目录层次将 **home** 目录分类。例如, 在大学中, 可以将 **home** 目录分别归置在名为 **undergrad**、**grad**、**professors** 和 **staff** 的子目录中。

唯一一个 **home** 目录不在 **/home** 目录下的用户标识是超级用户(**root**)。因为管理员必须总是能够控制系统, 所以超级用户的 **home** 目录必须总是可用的, 即使在系统启动过程中, 或者以单用户模式(参见第 6 章)运行时。在许多系统上, **/home** 目录属于辅助文件系统,

所以在挂载之前不可用。另一方面，`/root` 目录属于根文件系统，因此总是可用的。

每次登录时，环境变量 `HOME` 被设置为自己 `home` 目录的名称。因此，一种显示 `home` 目录名称的方式就是使用 `echo` 程序显示 `HOME` 变量的值：

```
echo $HOME
```

(`echo` 程序只是简单地显示其参数的值，该程序在第 12 章中与环境变量一起讨论。)

作为快捷方式，符号 `~`(波浪号)可以用作 `home` 目录的缩写。例如，通过使用下述命令也可以显示 `home` 目录的名称：

```
echo ~
```

无论 `home` 目录的名称是什么，最重要的是该目录是可以根据自己意愿使用的目录。在自己的 `home` 目录中，第一件事情就是创建一个 `bin` 子目录来存储自己的私人程序和 shell 脚本，然后再将这个目录的名称(例如 `/home/harley/bin`)放在自己的搜索路径中。

(搜索路径是一串目录，存储在 `PATH` 环境变量中。每当输入不是 shell 内置程序的程序名称时，Unix 就会在搜索路径所指定的目录中进行搜索，查找合适的程序来执行。详情请参见第 13 章。)

图 23-9 示范了一个基于 `home` 目录 `/home/harley` 的典型目录结构。这个 `home` 目录有 3 个子目录 `bin`、`essays` 和 `games`。`essays` 目录中又有 3 个子目录 `history`、`literature` 和 `surfing`。图中没有列出这些目录包含的文件。从第 24 章可以看出，生成和移除子目录非常容易。因此，在需要变化时，扩大或剪除目录树是一件非常简单的事情。

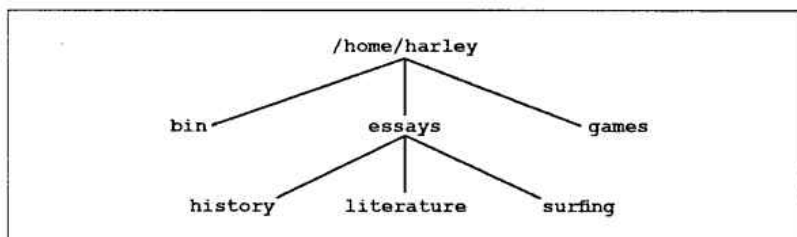


图 23-9 一个典型的基于 `home` 目录的树结构

每个用户标识都被赋予一个 `home` 目录。根据文件系统层次结构标准，`home` 目录应该位于 `/home` 目录中，但是各种系统有所不同(参见正文)。

在 `home` 目录中，可以根据需求创建和移除子目录。本例示范了一个典型的 `home` 目录，其中有 3 个子目录。在这 3 个子目录中，有 1 个子目录自己又拥有 3 个子目录。所有 6 个子目录都包含有文件，但是图中没有列出它们。

`/home` 目录属于文件系统层次结构标准，并且在 Linux 系统中广泛使用。但是，如果使用的是另一种类型的 Unix，则可能发现 `home` 目录位于不同的位置上。经典的设置，也是使用过很多年的方法是 `将 home 目录放在 /usr 目录中`。例如，用户标识 `harley` 的 `home` 目录将是 `/usr/harley`。还有一些系统使用的是 `/u`、`/user`(有一个“e”)、`/var/home` 或者 `/export/home`。出于参考目的，下面列举了一些可能在不同系统上看到的 `home` 目录位置：

```

/usr/harley
/u/harley
/user/harley

```

```
/var/home/harley
/export/home/harley
```

在大型系统上，特别是那些文件存储在网络中的系统上，home 目录的准确位置更加复杂，它更取决于系统管理员决定如何组织文件系统。例如，我在一台计算机上拥有一个账号，其 home 目录是子-子-子-子-子目录：

```
/usr/local/psa/home/vhosts/harley
```

提示

在任意系统上，都可以通过下述命令之一查找 home 目录的位置：

```
echo $HOME
echo ~
```

23.17 虚拟文件系统

在本章中，我们已经讨论了大量的具体内容。对关心这一类事情的人来说，这些细节还已经算是有趣的了。但是，事情远未结束。Unix 文件系统由少数几个非常聪明的人创建，并且多年以来，得到许多有经验的程序员和管理员的增强，最终使其不仅实用，而且非常漂亮。

在本节中，我希望大家不仅欣赏到文件系统的实用性，而且还欣赏到它的完美性。为此，我将解释如何将不同存储设备上的多个文件系统组合成一个大型的树形结构。

在本章前面，我解释过每个存储设备都拥有自己的本地文件系统，其中的目录和子目录以标准 Unix 方式组织成树。在访问这样的文件系统之前，该文件系统必须连接到主文件系统中，这一过程称为挂载。从技术上讲，我们通过将文件系统的根目录连接到一个挂载点(主文件系统中的目录)来挂载一个文件系统。

我希望您注意到，当谈论这些思想时，我们以两种不同的方式使用单词“文件系统”，不要搞混了。首先是“Unix 文件系统”，这个大型的、无所不包的结构包含整个系统中的每个文件和每个目录。其次是小型的，位于不同存储设备上的单个“设备文件系统”。通过将小型的设备文件系统连接到一个大型的结构可以创建 Unix 文件系统。

为了解释 Unix 文件系统的运转方式，我们从回答下述问题开始：当系统引导时会发生什么事情呢？当打开计算机电源时，会发生一串复杂的事件，这称为引导过程(详见第 2 章中的描述)。在开机自检之后，一个称为引导加载程序的程序接管计算机，从引导设备中读取数据，从而将操作系统加载到内存中。在大多数情况下，引导设备是本地硬盘驱动器上的一个分区。但是，它也可以是网络设备、CD、闪存驱动器等。

在引导设备的数据中有初始的 Unix 文件系统，称为**根文件系统**。根文件系统自动挂载，其中存放着启动 Unix 所需的全部程序和数据文件。根文件系统中还包含有系统出现问题时系统管理员需要使用的工具。因此，根文件系统至少包含下述目录(本章前面已经讨论过)：

```
/bin /boot /dev /etc /lib /root /sbin /tmp
```

一旦挂载了根文件系统，内核也启动完毕，就会自动挂载其他设备文件系统。此类文

件系统的信息存放在配置文件 `/etc/fstab`^{*} 中, 该文件可以由系统管理员修改(这一名称代表“file system table”, 即文件系统表)要查看系统上的这个文件, 可以使用命令:

```
less /etc/fstab
```

根文件系统总是存储在引导设备上。除此之外, 有 3 个其他文件系统可能位于单独的设备上: **usr**、**var** 和 **home**。如果这些文件系统都位于自己的设备上, 那么通过将它们附加在合适的子目录上可以使它们连接到 Unix 文件系统上。**usr** 文件系统挂载在 **/usr** 上; **var** 文件系统挂载在 **/var** 上; 等等。这都是自动完成的, 因此当看到登录提示时, 所有的东西都已经挂载, Unix 文件系统已经启动并且正在运行。

每个设备使用一种适合该设备类型的文件系统。硬盘驱动器上的分区使用适合硬盘驱动器的文件系统; CD-ROM 使用适合于 CD-ROM 的文件系统; 等等。可以想象, 根据设备类型的不同, 文件系统在读取和写入数据的细节方面存在巨大的差别。根据文件系统是本地的(在自己的计算机上)还是远程的(在网络上), 这些细节也存在巨大的差别。最后, 一些文件系统, 例如 `proc` 文件的 **procfs**, 使用的是伪文件, 这些文件不驻留于存储设备。出于参考目的, 图 23-10 列举了一些最常使用的文件系统。

基于磁盘的文件系统	
ext3	第 3 代扩展文件系统(Linux)
ext4	第 4 代扩展文件系统(Linux)
FAT32	32 位文件分配表文件系统(Microsoft Windows)
HFS+	层次式文件系统(Macintosh)
ISO 9660	ISO 9660 标准文件系统(CD-ROM)
NTFS	NT 文件系统(Microsoft Windows)
UDF	通用磁盘格式文件系统(可重写 CD 和 DVD)
UFS2	Unix 文件系统(BSD、Solaris)
网络文件系统	
NFS	网络文件系统(广泛使用)
SMB	服务器信息块(Windows 网络)
特殊用途文件系统	
devpts	伪终端的设备界面(PTY)
procfs	<code>proc</code> 文件系统
sysfs	系统数据文件系统(设备与驱动器)
tmpfs	临时存储文件系统

图 23-10 最常见的文件系统

出于参考目的, 这里列举了在使用 Unix 或 Linux 系统时将遇到的最常见的文件系统。基于磁盘的文件系统在硬盘、CD、DVD 或其他设备上存储数据; 网络文件系统支持通过网络共享资源; 特殊用途文件系统提供对系统资源(例如伪文件)的访问。

各种文件系统之间存在的巨大差别引出一个重要的问题。考虑下述两条 **cp**(copy, 复制)命令:

^{*} 在 Solaris 上, 该文件命名为 `/etc/vfstab`。

```
cp /media/cd/essays/freddy-the-pig /home/harley/essays
cp /proc/cpuinfo /home/harley/
```

我们将在第 25 章中讨论 `cp` 命令。现在，我只希望您知道第一条命令将一个文件从 CD 上的一个目录复制到硬盘分区上的一个目录中。第二条命令将伪文件中的信息(由内核生成)复制到硬盘分区上的一个文件中。在两个例子中，您只输入了一条简单的命令，那么是谁来处理这些细节呢？

细节由一个称为虚拟文件系统(virtual file system, VFS)的特殊工具进行处理。VFS 是 API(application program interface, 应用程序界面)，充当程序和各种文件系统的中间人。每当程序需要 I/O 操作时，它就向虚拟文件系统发送一个请求。VFS 定位合适的文件系统，通知设备驱动程序执行 I/O 与之进行通信。通过这种方式，VFS 允许用户和程序以一个单独的、一致的树形结构(Unix 文件系统)协调工作，即便数据实际上来源于各种独立的异构文件系统。

在第一个例子中，数据必须从 CD 读取。`cp` 程序发送一个读请求，该请求由虚拟文件系统处理。VFS 将它自己的请求发送给 CD 文件系统。CD 文件系统向 CD 设备驱动程序发送合适的命令以读取数据。通过这种方式，用户和程序都不需要知道任何细节。就用户的感受而言，Unix 文件系统就如他想象的那样存在，并且按照希望的方式运转。

您能看到完美性吗？在每个文件操作的一端，虚拟文件系统以用户语言与用户沟通。在另一端，VFS 以设备文件系统的语言与各种设备文件系统沟通。最终，用户和程序能够与任何文件系统交互，而且不必与文件系统直接沟通。

现在考虑另一个问题。每当开发新类型的文件系统时(假设为新设备开发)，如何使它适合于 Unix 呢？答案在概念上讲非常简单。所有新设备的开发人员只需教新文件系统说“VFS”语言，就能够使这个新文件系统加入到 Unix 世界中来，并且无缝地集成到其中。

下面是另一个完美之处：无论何时学习 Unix——35 年前还是 35 分钟之前，Unix 文件系统都有相同的外观和相同的运转方式。此外，新的设备和更好的文件系统多年以来一直在不停地开发，它们都平滑简易地集成到 Unix 中来。这就是为什么在学生带着彩色长念珠抗议战争的时代开发的操作系统，在学生拿着手机抗议战争的时代仍然能够很好地运转的原因。

那么未来又会如何呢？我们不知道未来会开发出什么稀奇古怪的新设备，以及有什么样的信息源可用。毕竟，对于技术，没有人可以保证什么。但是，我可以保证的是，无论出现什么新技术，它都可以与 Unix 协调工作。而且我还可以保证，多年之后，您还会教自己的孩子学习 Unix*。

23.18 练习

1. 复习题

1. 什么是 Unix 文件？3 种主要的文件类型各是什么？请描述每一种文件类型。
2. 解释文本文件和二进制文件之间的不同。请为每种文件举 3 个例子。
3. 什么是文件系统层次结构标准(FHS)？在 FHS 中，简短地描述下述目录的内容(1)

* 所以最好保存好本书。

根目录(/); (2)顶级目录/bin、/boot、/dev、/etc、/home、/lib、/sbin、/tmp、/usr 和/var。

4. 在 FHS 中, 哪些目录中包含通用的程序? 哪些目录包含系统管理程序?

5. 什么是 home 目录? 在 FHS 中, 到什么地方查找 home 目录? 唯一一个与其他 home 目录不在同一个位置的用户标识是什么? 为什么?

假设您的用户标识是 **weedly**, 而您是一所大型大学的本科生。请给出两个您的 home 目录的可能名称。

2. 应用题

1. 下述命令将列举根目录中的所有子目录:

```
ls -F / | grep '/'
```

使用该命令查看系统中顶级目录的名称。将结果与文件系统层次结构标准的基本布局进行比较。它们之间有什么不同?

2. 正如第 24 章中将讨论的, 可以使用 **cd** 命令从一个目录切换到另一个目录, 或者使用 **ls** 列举目录的内容。例如, 为了切换到 **/bin** 目录并列举其内容, 可以使用:

```
cd /bin; ls
```

探索自己的系统, 并查找下述文件的存放位置:

- 用户的 home 目录
- 通用程序(Unix 实用工具)
- 系统管理程序
- 特殊文件
- 配置文件
- 说明书页
- 内核
- 引导系统所需的文件

提示: 可以使用 **whereis** 程序(参见第 25 章)

3. 输入下述命令:

```
cp /dev/tty tempfile
```

键入几行文本, 然后按[^]D。刚才做的是什? 解释它的运转机理。

提示: 为了自己动手清除磁盘, 应该输入下述命令移除(删除)文件 **tempfile**:

```
rm tempfile
```

3. 思考题

1. Unix 以一种通用的方式定义“文件”。列举这种系统的 3 个优点, 再列举 3 个缺点。

2. Unix 系统在必须遵循文件系统层次结构标准的程度上没有一致性。一些系统坚持遵循该标准, 而其他系统却不这样做。那么, 要求所有的 Unix 系统都使用相同的基本文件系统层次结构是好还是坏呢? 讨论各自的优缺点。

目录操作

本章是介绍 Unix 文件系统的 3 章中的第 2 章。在第 23 章中，我们从总体上讨论了文件系统：文件系统如何以目录和子目录的形式组织成树形层次结构，如何使用文件系统的各个部分，以及使用 Unix 时将遇到的各种文件类型。

在整个层次结构中，每个用户都被赋予一个 **home** 目录，用户可以根据自己的需求进行组织。为了处理树中自己的那一部分，以及整个文件系统，我们需要能够轻松快速地从一個目录导航到另一个目录。另外，还需要能够根据需求通过创建、删除、移动及重命名子目录，来管理自己的文件。最后，需要能够查看各种目录的内部，从而可以操作目录中的文件和子目录。

在本章中，我们将学习目录操作所需的全部基本技能。在第 25 章中，我们通过介绍操作常规文件的命令结束对文件系统讨论。

24.1 路径名与工作目录

在第 23 章中，我们讨论了如何书写文件的完整名称。首先是一个 **/**(斜线)字符，该字符代表根目录。然后是到达该文件途经的所有目录名，每个目录名后面跟一个 **/**，最后是文件名。例如：

```
/usr/share/dict/words
```

在这个例子中，文件 **words** 位于 **dict** 目录中，**dict** 目录位于 **share** 目录中，**share** 目录又位于 **usr** 目录中，而 **usr** 目录又位于根目录中。

当以这种方式书写文件的名称时，我们描述了从根目录到正在讨论的文件所经过的目录树的路径。这样做时，我们指定了一个目录序列，目录间用 **/** 字符分隔开。该描述称为**路径名**或**路径**。上面示范的例子就是一个路径名。

如果路径名的最后一部分是普通文件的名称，我们就称它为**文件名**，或者**基名**(**basename**)，但是这个称呼不太常用。在我们的例子中，**words** 就是文件名。

下面再举一个路径名的例子。假设您的用户标识是 **harley**，**home** 目录是 **/home/harley**(参见第 23 章)。您希望使用 **vi** 文本编辑器(参见第 22 章)编辑文件 **memo**。为

了启动 vi，输入下述命令：

```
vi /home/harley/memo
```

过了一会，您决定编辑另一个文件 **document**，所以输入：

```
vi /home/harley/document
```

在这些例子中，路径名分别是：

```
/home/harley/memo
/home/harley/document
```

文件名分别是：

```
memo
document
```

可见，每次希望访问文件时都键入完整的路径名非常麻烦，且易于出错。为了方便起见，Unix 允许在某一时刻指定一个目录作为**工作目录**(也称为**当前目录**)。每当希望使用工作目录中的文件时，无需指定整个路径，只需键入文件名即可。例如，如果您已经告诉 Unix 自己正工作在 **/home/harley** 目录中(现在还不必深入讨论细节)，那么下述命令是等价的：

```
vi /home/harley/memo
vi memo
```

基本规则如下：当使用以/开头的名称时，Unix 假定这个名称是一个完整的路径名，从根目录开始。第一条命令就是这种情况。当只使用文件名时，Unix 假定引用的是工作目录中的文件。第二条命令就是这种情况(稍加练习，就会理解这一规则)。

每次登录时，Unix 自动将工作目录设置为 **home** 目录^{*}，而 **home** 目录正是开始工作的方便位置。在工作过程中，可以在需要使用 **cd**(change directory, 改变目录)命令改变工作目录，我们将在本章后面对此展开讨论。在工作会话期间，根据正在做什么事情，时常需要改变工作目录。但是，工作目录与在什么地方结束会话没有关系。下一次登录时，将再次以 **home** 目录作为工作目录。

下面是我希望大家思考的方式。将 Unix 文件系统想象成一棵大树。树干就是根目录，其他目录都是树枝。例如，目录 **/home** 和 **/bin** 都是根的分支。目录 **/home/harley** 是 **/home** 的分支。在任何时候，您都要位于一个树枝上，而这个树枝就是工作目录。

在登录时，所处的位置是表示 **home** 目录的树枝。为了移动到另一个树枝上，只需改变工作目录。因此，可以将 **cd** 命令看作一个具有魔力的地毯，可以瞬间从一个树枝移动到另一个树枝上的地毯。

24.2 绝对路径名与相对路径名

路径名或路径通过列举由/(斜线)字符分隔开的目录序列描述文件树中的一个位置。如

^{*} Unix 如何知道用户的 **home** 目录的名称呢？每个用户标识的 **home** 目录的路径名都存储在 Unix 口令文件 **/etc/passwd** 中，详情请参见第 11 章中的描述。

果目录序列从根目录开始，则称之为**绝对路径名**(absolute pathname)。如果目录序列从工作目录开始，则称之为**相对路径名**(relative pathname)。

我们使用图 24-1 中所示的目录树来描述绝对路径名和相对路径名之间的区别。这棵树显示的子目录属于用户标识 **harley**，该用户标识的 **home** 目录是 **/home/harley**(第 4 章中讲过，Unix 文件由用户标识拥有，而不是用户)。

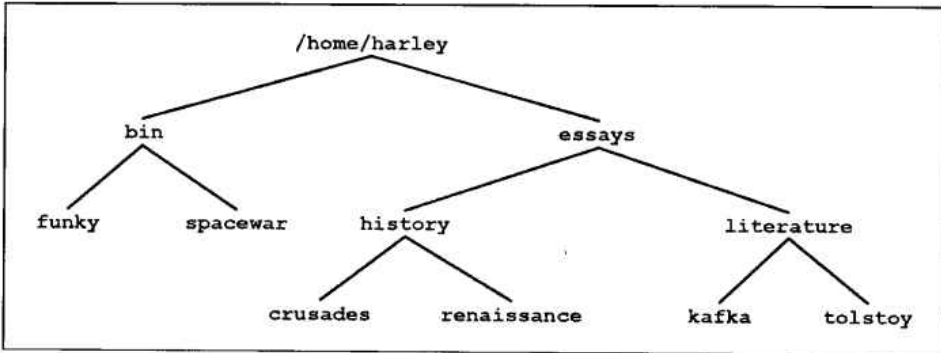


图 24-1 目录树示例

在用户标识 **harley** 的 **home** 目录中，有两个子目录：**bin** 和 **essays**。第一个子目录中有两个普通文件。第二个子目录中有两个子目录，每个子目录又包含有两个普通文件。正文中将使用这个小型的树形结构描述绝对路径名和相对路径名之间的区别。

在 **home** 目录中，有两个子目录 **bin** 和 **essays**。按照 Unix 的传统，**bin** 目录中的文件包含的是可执行程序 and 脚本(参见第 23 章)。在这个例子中，**bin** 中有两个这样的程序 **funky** 和 **spacewar**。**essays** 子目录中包含两个子目录 **history** 和 **literature**。每个子目录中又包含两个普通文件。当用户标识 **harley** 登录时，工作目录就被自动地设置为其 **home** 目录 **/home/harley**。下面看一看如何指定各种文件的名称。

假设我们希望使用一条命令，该命令需要引用 **bin** 目录。Unix 假定任何以 **/** 开头的名称都是绝对路径名。也就是说，它从根目录开始，显示了文件的完整路径。如果名称没有以 **/** 开头，那么 Unix 假定这个名称是相对路径名。也就是说，它描述了一个从工作目录开始的路径。

我们以两种方式引用 **bin** 目录。首先是绝对路径名：

```
/home/harley/bin
```

另外，因为工作目录是 **/home/harley**，所以使用相对路径名要更简单些：

```
bin
```

下面再举一个使用同一个工作目录的例子。我们希望输入一条命令，在这条命令中，需要指定 **literature** 目录中 **tolstoy** 文件的名称。绝对路径名是：

```
/home/harley/essays/literature/tolstoy
```

同样，相对路径名比较短：

```
essays/literature/tolstoy
```

下面举最后一个例子。假设我们希望大量地使用文件 **kafka** 和 **tolstoy**，使用绝对路径名来引用这两个文件不怎么方便：

```
/home/harley/essays/literature/kafka
/home/harley/essays/literature/tolstoy
```

但是，使用相对路径名也只相对方便了一点：

```
essays/literature/kafka
essays/literature/tolstoy
```

最好的办法就是改变工作目录到：

```
/home/harley/essays/literature
```

(稍后将示范如何去做。)一旦改变了工作目录，我们就可以更简捷地引用这两个文件：

```
kafka
tolstoy
```

工作目录是各种操作的基础，在需要时可以进行改变。在登录时，工作目录就是 **home** 目录，但是当需要时，可以随时将其改变到自己希望的目录。这里的思想就是选择合适的工作目录，从而使文件名尽可能简单，以易于键入。

本书自始至终有许多例子都采取类似于下面的方式使用文件名：

```
vi kafka
```

现在您已经理解了，在这些情况中，实际上使用的是相对路径名。在这个例子中，命令使用工作目录中的文件 **kafka** 启动 **vi** 文本编辑器。当然，在需要时，使用完整的路径名永远是允许的：

```
vi /home/harley/essays/literature/kafka
```

每当需要使用的程序要求指定文件的名称时，我希望您记住这一思想——在这些情况中，既可以使用绝对路径名，也可以使用相对路径名。

提示

当需要使用普通文件名时，Unix 的基本规则就是可以使用路径名代替。

24.3 3 种便利的路径名缩写：...~

Unix 提供了 3 种便利的路径名缩写。第一种缩写是连续两个点号，其发音是“dot-dot”：

```
..
```

当在路径名中使用 **..** 时，它指的是父目录。

为了说明其使用方法，我们还以图 24-1 中的目录树为例。在 `home` 目录/`home/harley` 中，有两个子目录 `bin` 和 `essays`。`bin` 目录中包含两个文件。`essays` 子目录自己也包含两个子目录 `history` 和 `literature`，每个子目录中包含两个文件。假设将工作目录设置为：

```
/home/harley/essays/literature
```

(这样做时，需要使用 `cd` 命令，`cd` 命令在本章后面讨论。)

一旦以这种方式改变了工作目录，就可以以 `kafka` 和 `tolstoy` 直接引用该目录中的两个文件(使用相对路径名)。此时，`..` 引用的是父目录，也就是：

```
/home/harley/essays
```

假设希望引用 `history` 目录中的文件 `crusades`。一种方法就是键入整个绝对路径名：

```
/home/harley/essays/history/crusades
```

另一种较简单的方法就是使用缩写`..`代表父目录：

```
../history/crusades
```

当使用`..`时，它相当于父目录的名称，所以上述路径名等价于绝对路径名。

使用`..`多次可以向“上”移动多个层次。例如，从同一个工作目录出发，假设希望引用 `bin` 目录。使用绝对路径名时，命令为：

```
/home/harley/bin
```

另外，也可以使用`..`缩写两次：

```
../../bin
```

第一个父目录是：

```
/home/harley/essays
```

第二个父目录(祖父目录)是：

```
/home/harley
```

下面再举一个例子。假设希望引用 `bin` 目录中的 `funky` 文件，绝对路径名为：

```
/home/harley/bin/funky
```

从同一个工作目录出发，还可以使用：

```
../../bin/funky
```

下面举最后一个例子，这个例子有点极端(慢慢地读，确保理解这个例子)。假设现在的工作目录是：

```
/home/harley/essays/literature
```

为了引用整个文件系统的根目录，可以使用`..` 4 次：

```
../..../..
```

同样,可以采用如下方式引用/etc 目录:

```
../..../..../etc
```

当然,或许您永远也不会使用这些例子,因为键入/和/etc 更容易些。当希望指定的目录靠近工作目录时,..**缩写**最为有用,而且无需实际改变工作目录。

第二个路径名缩写是一个单点号,通常称之为“点”:

```
.
```

一个单独的.指的是工作目录本身。例如,假设当前工作目录是:

```
/home/harley/essays/literature
```

那么下面 3 种表达式都引用同一个文件:

```
/home/harley/essays/literature/kafka
./kafka
kafka
```

当然,键入一个.要比键入工作目录的完整名称容易许多。但是,正如所见,实际上无需指定任何目录名称。只要名称不以/开头,Unix 就假定该路径名是相对于工作目录的。这一原则非常重要,因此我希望以提示的形式强调这一原则:

提示

任何不以/开头的路径名都被认为相对于工作目录。

现在,您可能会奇怪,为什么需要使用一个单独的.缩写呢? 在一些情形中,必须指定绝对路径名。这时,可以使用.缩写表示工作目录的名称。使用该方式避免键入长的路径名,不仅仅是出于懒惰(尽管这是一个不错的主意),更是因为这样不易于出现拼写错误(我确信您现在应该知道,当键入 Unix 命令时,非常容易出现拼写错误)。

下面举例说明。假设您在编写一个程序 **plugh**(这里不解释该程序的用途,大家可以自由想象)。该程序位于目录/home/harley/adventure 中,此时,这个目录就是您的工作目录。通常,可以通过输入程序的名称来运行程序:

```
plugh
```

但是,Unix 只能运行它能找到的程序。在大多数情况下,这意味着存放程序的文件应该位于搜索路径(参见第 13 章)的一个目录中。在我们的例子中,包含该程序的目录不在搜索路径中。但是,如果指定绝对路径名的话,那么 Unix 总是能够找到程序并运行它。因此,可以通过键入下述命令运行 **plugh** 程序:

```
/home/harley/adventure/plugh
```

因为程序位于工作目录中,所以有一种更简单的方法,即使用.缩写:

```
./plugh
```

`..`和`.`都是缩写，一定要确保理解这一点。在我们描述的例子中，当名称以`..`或`.`开头时，实际上指定的是一个完整的路径名。Unix 只是在帮助命令的键入。

第三种路径名缩写是`~`(波浪号)。可以在路径名的开头使用这个字符代表 home 目录。例如，为了使用 `ls` 程序列举 home 目录中的全部文件名，可以使用：

```
ls ~
```

为了列举 home 目录的子目录 `bin` 中的文件，可以使用：

```
ls ~/bin
```

为了引用另一个用户标识的 home 目录，可以使用一个`~`字符，后面跟着用户标识。例如，为了列举用户标识 `weedly` 的 home 目录中的文件，可以使用：

```
ls ~weedly
```

假设 `weedly` 自己拥有一个 `bin` 目录，在该 `bin` 目录中，有一个 `mouse` 程序。为了运行这个程序，必须键入绝对路径名。您有两种选择：

```
/home/weedly/bin/mouse  
~weedly/bin/mouse
```

最后几个例子引出了一个重要的问题。任何用户都能够查看其他人的文件，以及运行其他人的程序吗？用户能够改变其他人的文件吗？

答案是所有文件(包括目录)都拥有“权限”。文件的权限表明谁可以查看及修改该文件。在许多系统上，默认设置是允许用户查看其他人的文件，但是不允许修改或者运行文件。尽管如此，由于文件权限归文件的属主控制，因此，每个 Unix 用户可以根据自己的需要限制或者允许他人访问自己的文件。我们将在第 25 章中讨论这些问题。

技术提示

在指定路径名时有 3 种标准的缩写可以使用：`.`(当前目录)、`..`(父目录)和`~`(home 目录)。尽管它们看上去相似，但是它们并没有采用相同的方式实现。

名称`.`和`..`是实际目录条目，由文件系统自动创建。系统中的每个目录都包含有这两个条目。

名称`~`是 shell 提供的一种抽象概念，是为了方便引用 home 目录而提供的。

24.4 在目录树中移动：cd、pwd

在显示工作目录的名称时，可以使用 `pwd`(print working directory, 显示工作目录)命令。该命令的语法比较简单：

```
pwd
```

为了切换工作目录，可以使用 `cd`(change directory, 改变目录)命令。该命令的语法为：


```
cd [-LP] [directory | -]
```

其中 *directory* 是希望切换到的目录的名称。

如果在输入该命令时没有指定目录名称,那么 **cd** 命令在默认情况下,将切换到 **home** 目录。如果在输入该命令时使用 **-** 代替目录名称,那么 **cd** 将切换到前一目录。**-L** 和 **-P** 选项与符号链接相关,我们将在第 25 章中讨论符号链接。

通常,当 Unix 名称短小且其中没有元音时,发音时就拼成单独的字母。例如, **ls** 命令发音为“L-S”。同理, **pwd** 和 **cd** 命令的发音为“P-W-D”和“C-D”。

在所有的 Unix 工具中, **cd** 和 **pwd** 是最有用的工具。您将会大量地使用它们,所以一定要仔细地阅读本节。下面举一些例子,示范如何使用 **cd** 命令。在自己练习的过程中,记着时不时地使用一下 **pwd** 命令,以检查自己的位置。

为了将工作目录切换到 **/home/harley/essays**, 可以使用:

```
cd /home/harley/essays
```

为了切换到 **/bin**, 使用:

```
cd /bin
```

为了切换到 **/**(根目录), 使用:

```
cd /
```

为了方便起见,可以使用相对路径名以及缩写。例如,假设您的当前工作目录是 **/home/harley**。在这个目录中,有两个子目录: **bin** 和 **essays**。为了切换到 **bin**(也就是指 **/home/harley/bin**), 只需输入:

```
cd bin
```

因为目录名称 **bin** 没有以 **/** 开头,所以 Unix 认为它是一个基于当前工作目录的相对路径名。下面再举一个例子。再一次假设您的工作目录是 **/home/harley**, 而这一次您希望切换到:

```
/home/harley/essays/history
```

通过使用相对路径名,可以输入:

```
cd essays/history
```

当使用 **cd** 命令时,如果不指定目录名称,那么 **cd** 命令将把工作目录切换到 **home** 目录:

```
cd
```

在探索文件系统的过程中,如果您处于文件系统中一个较远的分支上,而自己在目录树上又迷失方向,则通过这种方式使用 **cd** 命令,您就可以快速地返回到 **home** 目录中。*

* 还有一种选择,如果您碰巧穿着红宝石拖鞋,则可以碰三下脚后跟,并重复说“没有什么地方比家好”,就可以实现自己的梦想。

例如，假设您的工作目录是 `/etc/local/programs`，而您现在希望移动到自己的 `home` 目录的 `bin` 目录中，只需输入：

```
cd
cd bin
```

第一条命令切换到 `home` 目录。第二条命令切换到 `home` 目录中的 `bin` 目录。为了使其更方便，前面讲过，可以在同一个命令行上输入多条命令，并用分号将命令分隔开(参见第 10 章)。因此，无论位于文件系统的什么地方，都可以通过输入下述命令移动到自己的 `bin` 目录中：

```
cd; cd bin
```

下面再举一些例子，示范如何使用前面讨论的两个标准路径名缩写。首先从 `..` 开始，该缩写代表父目录。下面，假设您现在的工作目录是：

```
/home/harley/essays/history
```

为了切换到父目录 `/home/harley/essays`，只需在目录树中向上移动一层：

```
cd ..
```

从原始的工作目录开始，可以使用下述命令切换到 `/home/harley/essays/literature`：

```
cd ../literature
```

为了向上移动不止一层，可以多次使用 `..` 缩写。例如，从原始工作目录开始，可以使用下述命令切换到 `/home/harley/bin` 目录：

```
cd ../../bin
```

问题：如果在根目录中输入下述命令，会发生什么情况？

```
cd ..
```

答案：不发生任何事情。您的工作目录不会发生改变，而且也不会看到错误消息。为什么会这样呢？因为 `Unix` 认为根目录的父目录就是根目录自身。^{*}例如，尽管看上去十分奇怪，但是下述两个路径名都指向同一个文件：

```
/etc/passwd
../../../../etc/passwd
```

另外一个有用的缩写是 `~`(波浪号)，正如前面讨论的，它代表 `home` 目录的名称。因此，下述两条命令行的结果相同：它们都将工作目录设置为 `home` 目录中的 `bin` 目录：

```
cd; cd bin
cd ~/bin
```

第一条命令分两步完成改变；第二条命令一步就完成改变。

有时候，可能发现自己要在两个目录中来回切换。在这种情况下，`cd` 拥有一个特殊的

^{*} 一个拥有重要理论价值的假设。

缩写,可以使这种类型的切换十分容易。如果使用`-`代替目录名称,那么 `cd` 将切换到上一次访问的目录。同时,`cd` 将显示新目录的名称,从而使您知道自己位于何处。

下面举例说明,您自己可以试一试。首先,使用 `cd` 切换到`/etc` 目录,然后使用 `pwd` 命令确认已经切换到`/etc` 目录:

```
cd /etc; pwd
```

现在切换到`/usr/bin`:

```
cd /usr/bin; pwd
```

最后,输入 `cd` 和一个`-`字符:

```
cd -
```

现在又返回到了`/etc`。

提示

在任何时候,都可以使用 `pwd` 命令显示工作目录的名称,从而查看自己在文件系统树中的位置。此外还有另外两种方法。

第一,可以在 shell 提示中显示工作目录的名称。在由一个目录切换到另一个目录时,shell 提示会自动更新,显示位于什么位置。详情请参见第 13 章。

第二,大多数基于 GUI 的终端窗口都在标题栏(位于窗口的顶部)中显示工作目录的名称。大家可以看看自己的系统是不是这种情况。

名称含义

`pwd`、`cd`

在第 3 章中,我们讨论了早期的 Unix 开发人员如何使用电传打字机终端在纸张上打印输出。多年以来,Unix 保留了单词“print, 打印”的使用传统,即该单词意味着“显示信息”。因此,虽然人们将工作目录的名称打印在纸张上已经过去了很长时间,名称 `pwd` 依旧代表“print working directory, 显示工作目录”。

如果和一群 Unix 极客待在一起,那么您会经常听到他们将 `cd` 作为动词使用(当他们这样做时,名称 `cd` 的发音是两个单独的字母“C-D”)。例如,有些人可能会说:“为了查找基本的 Unix 工具,只需 C-D 到`/bin` 目录进行查看。”这种方式类似于我们在前面使用的隐喻——想象自己坐在目录树的一个树枝上,使用 `cd` 可以从一个树枝移动到另一个树枝上,使用 `pwd` 可以提醒我们位于哪一个树枝上。

24.5 创建新目录: `mkdir`

创建目录时,可以使用 `mkdir` 程序。该程序的语法为:

```
mkdir [-p] directory...
```

其中 *directory* 是希望创建的目录的名称。

该程序的使用比较简单。只要遵循几条简单的规则，就可以自由地命名新目录。在第 25 章中，当讨论文件的命名时，我们再讨论这些规则(记住，正如第 23 章中解释的，目录其实就是文件)。基本而言，可以使用字母、数字以及没有特殊含义的标点符号作为文件的名称。但是，大多数时候，坚持只使用小写字母将使生活更轻松。

下面举例说明。为了在自己的工作目录中创建目录 **extra**，可以使用：

```
mkdir extra
```

当指定目录名称时，既可以使用绝对路径名，也可以使用相对路径名，同时还可以使用标准缩写。作为示例，假设您希望创建图 24-2 中的目录树(本章前面作为示例的目录)。在 **home** 目录中，希望创建两个子目录 **bin** 和 **essays**。在 **essays** 目录中，还要创建两个子目录 **history** 和 **literature**。

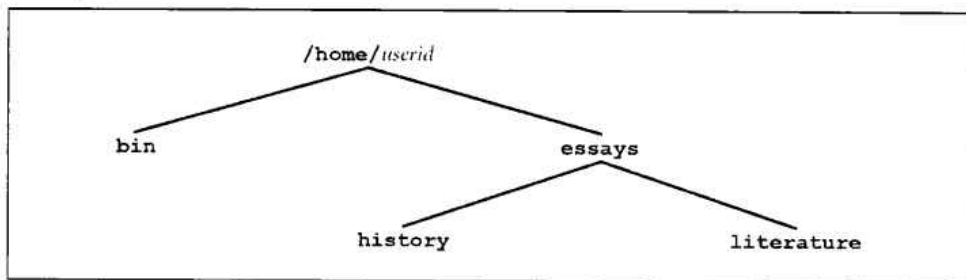


图 24-2 创建示例目录树

为了创建新目录，可以使用 **mkdir** 程序。这里是一个示例目录树，由示例 **mkdir** 命令创建(详情请参见正文)。这个树在 **home** 目录中包含两个子目录：**bin** 和 **essays**。**bin** 目录自己又包含两个子目录：**history** 和 **literature**。

首先，确保位于 **home** 目录中：

```
cd
```

现在，创建前两个子目录：

```
mkdir bin essays
```

接下来，进入 **essays** 目录，创建最后两个子目录：

```
cd essays
```

```
mkdir history literature
```

为了描述各种指定路径名的方式，我们首先看一下另外两种创建相同目录的方式。第一种可以不开 **home** 目录完成所有的事情：

```
cd
```

```
mkdir bin essays essays/history essays/literature
```

第一条命令切换到 **home** 目录。第二条命令指定全部 4 个目录的名称，这 4 个目录名称都相对于工作目录。在下述例子中，我们甚至不用切换到 **home** 目录：

```
mkdir ~/bin ~/essays ~/essays/history ~/essays/literature
```

记住, ~(波浪号)字符是 home 目录的缩写。

有时候, 使用..缩写来表示父目录非常方便。例如, 假设您已经切换到 **essays** 目录:

```
cd ~/essays
```

现在您决定在 **bin** 目录中创建一个子目录 **extra**。因为 **bin** 和 **essays** 目录拥有相同的父目录(home 目录), 所以可以使用:

```
mkdir ../bin/extra
```

在创建目录时, 要遵循 Unix 制定的两条合理的规则。第一, 在一个目录中, 不能以相同名称创建两个子目录。例如, 不能有两个名称都是 **~/essays/history** 的目录(否则, 如何区分它们呢?)但是, 如果在不同的父目录中, 则两个目录可以拥有相同的名称。例如:

```
~/essays/history
~/homework/history
```

第二条规则就是, 默认情况下, 如果父目录不存在, 则不能创建子目录。例如, 除非已经创建了 **~/homework** 目录, 否则就不能创建 **~/homework/history** 目录。当在一条命令中指定不止一个目录时, **mkdir** 将按指定的顺序创建各个目录。因此, 下述命令是可行的, 因为您告诉 **mkdir**, 目录 **homework** 在目录 **history** 之前创建:

```
mkdir ~/homework ~/homework/history
```

但是, 下述命令将不可执行, 因为子目录不能在父目录之前创建:

```
mkdir ~/homework/history ~/homework
```

前面将文件系统比作一棵树, 树的主干是根目录, 每个树枝是一个子目录。这样, 这两条规则就可以描述为:

- (1) 不能创建两个相同的树枝。
- (2) 不能创建一个没有长在树上的新树枝。

为了方便起见, 可以使用 **-p**(make parent, 创建父目录)选项忽略第二条限制。这将告诉 **mkdir** 自动创建所有需要的父目录。例如, 假设您正在研究早期罗马人如何使用 Unix, 需要创建下述目录结构来存放文件:

```
~/essays/history/roman/unix/research
```

除非 **unix** 目录已经存在, 否则无法创建 **research** 目录; 如果不创建 **roman** 目录, 那么也没有办法创建 **unix** 目录; 依此类推。因此, 如果该路径上的目录一个都不存在, 则必须使用下述 5 条命令创建完整的结构:

```
mkdir ~/essays
mkdir ~/essays/history
mkdir ~/essays/history/roman
mkdir ~/essays/history/roman/unix
mkdir ~/essays/history/roman/unix/research
```

但是, 如果使用 **-p** 选项的话, 就可以使用一条命令创建全部结构:

```
mkdir -p ~/essays/history/roman/unix/research
```

提示

对于文件名来说, Unix 是对大小写敏感的, 这意味着 Unix 区分大写字母和小写字母(参见第 4 章)。例如, Unix 认为下述 3 个目录名称是不同的:

```
bin
Bin
BIN
```

我们将在第 25 章中讨论文件的命名。现在, 先给出下述建议。在命名目录时, 除非有很好的理由, 否则只使用小写字母。如果想分隔单词, 则可以使用 **-** 或 **_**, 例如:

```
backups-january
backups_january
```

如果将整个目录名称放在引号内, 则可以在目录名称中使用空格。但是, 最好不要这样做。这样做只会带来麻烦。

24.6 移除目录: **rmdir**

为了移除(删除)目录, 可以使用 **rmdir** 程序。该程序的语法比较简单:

```
rmdir [-p] directory...
```

其中 *directory* 是希望移除的目录的名称。

例如, 为了从工作目录中移除目录 **extra**, 可以使用:

```
rmdir extra
```

当使用 **rmdir** 时, 可以使用绝对路径名或相对路径名指定一个或多个目录名称。另外还可以使用标准缩写: **..** 代表父目录, **~** 代表 home 目录。

下面我们举几个例子, 使用的示例目录树是上一节中构建的目录树(参见图 24-2)。在 home 目录中, 有两个子目录 **bin** 和 **essays**。在 **essays** 目录中, 还有两个子目录 **history** 和 **literature**。假设您希望删除全部 4 个目录。完成这一任务的方法有好几种。第一种方法, 切换到 **essays** 目录中:

```
cd ~/essays
```

在这里, 可以删除两个子目录:

```
rmdir history literature
```

接下来, 切换到父目录(home 目录)中:

```
cd ..
```


移除两个主子目录:

```
rmdir bin essays
```

另一种方法是切换到 home 目录, 在一条命令中移除全部 4 个子目录:

```
cd  
rmdir essays/history essays/literature essays bin
```

最后, 还可以不移动到 home 目录中就完成全部工作:

```
rmdir ~/essays/history ~/essays/literature ~/essays ~/bin
```

当移除目录时, 要遵循 Unix 制定的两条规则。第一, 只有当一个目录为空的时候, 才能移除这个目录, 这是一种安全防护措施(如果一个目录中含有子目录或者文件, 那么这个目录就不是空的)。

下面举一个现实生活中的例子。一个星期日的深夜, 您在计算机实验室中使用 Linux 完成一个特殊的项目。您的 home 目录中有两个子目录 **data** 和 **olddata**。**data** 目录中包含有 100 个重要的文件。**olddata** 目录是空的。您决定移除 **olddata** 目录。但是, 在输入命令时, 一颗陨石打碎窗户, 正好击中坐在您身边的一个极客。在混乱中, 您不小心键入了:

```
rmdir data
```

幸运的是, Unix 已经准备好处理这样的偶发事件。您将看到这样的消息:

```
rmdir: data: Directory not empty
```

感谢 Unix 内置的安全防护措施, **data** 目录原封不动, 没有被删除。

如果想一次性移除一串空目录, 则可以使用 **-p(delete parent, 删除父目录)** 选项^{*}。这样将告诉 **rmdir** 自动移除所有需要移除的父目录。例如, 假设您拥有如下目录结构, 而且所有的目录都是空的。

```
~/essays/history/roman/unix/research
```

您希望移除全部 5 个子目录。如果没有 **-p** 选项, 则不得不从最内层的子目录开始, 逐级向上, 一次删除一个子目录:

```
cd  
rmdir essays/history/roman/unix/research  
rmdir essays/history/roman/unix  
rmdir essays/history/roman  
rmdir essays/history  
rmdir essays
```

有了 **-p** 选项, 就可以切换到 home 目录, 一次完成整个工作:

```
cd  
rmdir -p essays/history/roman/unix/research
```

如果目录不是空的, 那么这些命令都无法执行。正如前面所述, 这是一种安全防护措

^{*} 有时候称为 Oedipus 选项。

施。但是，在一些不常见的场合中，可能确实希望移除非空的目录。这时可以使用带有**-r**选项的**rm**命令。**rm -r**将移除所有的子目录及其内容，因此在使用时一定要特别小心。我们将在第 25 章中讨论 **rm** 程序，因此 **rm** 程序的细节将在那里解释。

刚才，我们提到 **rmdir** 强加了两条规则。第一条规则是不能移除非空的目录。第二条规则是不能删除工作目录和根目录之间的任何目录。例如，假设您的工作目录是：

```
/home/harley/essays/literature
```

那么您不能移除 **essays** 目录或者 **harley** 目录，因为它们位于工作目录和根目录之间。但是，可以移除下述目录：

```
/home/harley/essays/history
```

也就是说，您可以使用下述命令：

```
rmdir ../history
```

毕竟，**history** 目录没有位于您和根目录之间。如果希望移除 **essays** 目录，则必须首先移到离根目录近一点的位置，比如 **/home/harley**。接下来就可以移除这个目录：

```
cd /home/harley
rmdir essays/history essays/literature essays
```

问题：假设工作目录是 **/etc**，那么可以移除 **home** 目录中的一个子目录吗？

答案：可以，因为工作目录(**/etc**)没有位于根目录和希望移除的目录之间。

在记忆这一规则时，可以回想一下前面所说的文件系统与真实树的类比。树干是根目录。每个树枝都是一个子目录。在任何时候，您所处的那个树枝就是工作目录。移除一个目录就像锯掉树枝一样。移除目录的限制就是不能锯掉那个正支撑您的树枝。

提示

移除工作目录是有可能的。这就像砍掉您正坐在其上的树枝一样。或许 Unix 不允许这样做，但是确实可以这样做。

移除自己的工作目录只会给自己惹麻烦，所以不要这样做。*

24.7 移动或重命名目录：mv

为了移动或重命名目录，可以使用 **mv** 程序。该程序的语法为：

```
mv directory target
```

其中 **directory** 是希望移动或重命名的目录，**target** 是目标或新名称。

使用 **mv** 程序可以将目录从一个位置“移动”到另一个位置。如果新位置和原位置在同一个目录中，那么实际结果就是对原始目录重命名。这就是 **mv** 程序既可以移动目录又

* 尽管我告诉您不要这样做，但是我知道您会忍不住这样做——只是为了看看会发生什么事情。当这样做时，一定要使用一个临时子目录。不要移除 **home** 目录，否则会真的遇上麻烦。

可以重命名目录的原因。

下面示范几个例子。假设您在当前工作目录中有一个目录 **data**，您希望将这个目录的名称改成 **extra**。假定该目录中还没有命名为 **extra** 的目录，则可以使用下述命令：

```
mv data extra
```

原来名为 **data** 的目录现在被命名为 **extra** 了。

如果目标目录已经存在，那么 **mv** 将把原始目录移动到目标目录中去。例如，假设您有下述两个目录：

```
/home/harley/data  
/home/harley/storage
```

您希望将 **data** 目录移动到 **storage** 目录内。可以使用

```
mv /home/harley/data /home/harley/storage
```

当然，如果工作目录是 **/home/harley**，则可以将上述命令简化为：

```
mv data storage
```

data 目录的路径名现在为：

```
/home/harley/storage/data
```

当 **mv** 移动目录时，同时移动目录中的所有文件和子目录。例如，假设在移动之前，在 **data** 目录中有一个 **document** 文件。该文件的绝对路径名是：

```
/home/harley/data/document
```

移动之后，该文件的绝对路径名变为：

```
/home/harley/storage/data/document
```

如果在 **data** 目录之下有子目录——甚至有可能是一个完整的子树，那么 **mv** 程序同样也移动它们。因此，**mv** 程序具有 3 项用途：

- (1) 重命名目录
- (2) 移动目录
- (3) 移动整个目录树

mv 程序可以用来移动或重命名普通文件，就像移动或重命名目录一样。我们将在第 25 章中对此展开讨论。

24.8 使用目录栈：pushd、popd、dirs

在第 13 章中，我们解释过 Unix 命令有两种类型。外部命令是单独的程序。内置(或内部)命令直接由 shell 解释，并且只有在 shell 支持它们时才可用。本节准备示范如何使用 3 条内置的命令：**pushd**、**popd** 和 **dirs**。这些命令在 Bash、Tcsh 和 C-Shell 中都可用，但是在 Korn Shell 中不可用。

此时，我们已经讨论过目录的基本操作。您已经知道如何创建、删除、移动以及重命名目录，另外还知道如何改变工作目录以及显示工作目录的名称。我们还准备讨论非常重要的 **ls** 程序的各种变体。**ls** 程序可以显示目录的内容，查看目录中存放着什么。但是，在讨论 **ls** 程序之前，我们先花一点时间，展示一项高级技术，该技术可以帮助在目录树中任意移动。

在第 8 章中，我们讨论了数据结构的思想。数据结构是根据一组精确的规则存储和检索数据的实体。到目前为止，我们已经讨论了 3 种不同的数据结构：栈(第 8 章)、队列(第 23 章)和树(第 9 章和第 23 章)。这里，我们准备再次讨论栈，下面先快速地浏览一下有关栈的基本知识。

栈是一种数据结构，在这种数据结构中，每次只能存入和检索一个元素，在任何时候，下一个要检索的数据元素是最后一个存储进去的数据元素。有时候称这种安排为后进先出(last-in first-out, LIFO)。在存储数据元素时，我们称将数据压入(push)到栈中。最近一次压入的数据元素处在栈的顶(top)上。当从栈顶检索数据元素时，我们称之为从栈中弹出(pop)元素。非正式地讲，可以认为栈与自助餐厅中用弹簧加载的盘子叠相似。盘子压入到“栈”中，每次一个。当希望取盘子时，可以从栈顶弹出盘子，不能从其他位置取盘子。

shell 中也提供了一个类似的功能来存放目录名称。在任何时候都可以使用 **pushd** 命令将目录的名称压入到目录栈(directory stack)中。然后，可以使用 **popd** 命令从栈中弹出一个目录名称。此外，随时可以使用 **dirs** 命令显示栈的内容。这些命令的语法如下所示：

```
pushd [directory | +n]
popd [+n]
dirs [-c] [-l] [-v]
```

其中 *directory* 是目录的名称，*n* 是标识符。注意，当 **dirs** 使用选项时，选项必须单独指定，不能连在一起。例如，可以使用 **dirs -l -v**，但是不能使用 **dirs -lv**。

在本节中，我们将讨论这 3 条命令的最重要的使用方式。这些使用方式中有几种比较深奥的变体，在学习的过程中可以参考联机手册(查看 shell 内置命令的说明书页)。出于参考目的，图 24-3 中汇总了我们准备讨论的各条命令。

命令	动作
dirs	显示名称：home 目录显示为~
dirs -l	显示名称：home 目录显示为完整路径名
dirs -v	显示名称：每行一个，并且有数字标识符
pushd directory	改变工作目录：将 <i>directory</i> 压入到栈中
pushd +n	改变工作目录：将目录#n 移到栈顶
popd	改变工作目录：弹出栈顶
popd +n	从栈中移除目录#n
dirs -c	除当前工作目录外，移除栈中的全部目录

图 24-3 目录栈命令

目录栈是一种高级工具，允许维护一个目录列表，每当需要时，可以将工作目录改变为列表中的一个目录。

在任何时候，栈顶的目录名称就是当前的工作目录。改变这个名称就会自动地改变工作目录。同样，改变工作目录也会自动地改变栈顶目录的名称。向栈中压入目录名称，从栈中弹出目录名称或者选择一个目录名称移动到栈顶，这些都是控制栈的方式。每个操作都会改变栈顶，因此也会改变工作目录。详情请参见正文。

学习如何使用目录栈需要多加练习，不过它值得我们这样做。像拥有一张后台通行证就能够在滚石音乐会中东看西看一样，一旦掌握了这些细节，您就可以在文件系统中随意游走。而要点就是要记住一条简单的规则：

在任何时候，栈顶存放的就是工作目录的名称。

每当改变工作目录时，栈顶元素也会自动改变。反过来，每当改变栈顶元素时，工作目录也会自动改变(在继续下面的内容之前好好地想一想)。

下面举一些例子。首先使用 **cd** 命令切换到/etc 目录，使用 **pwd** 命令确保已经切换到/etc 目录：

```
cd /etc; pwd
```

现在显示栈的内容。要显示栈的内容，可以使用 **dirs** 命令加 **-v(verbose, 详细)** 选项。该选项告诉 **dirs** 在拥有行号的单独行上显示栈的每个元素。栈顶所在行号是#0。

```
dirs -v
```

输出为：

```
0 /etc
```

切换到/usr 目录，再次显示栈的内容：

```
cd /usr; dirs -v
```

注意栈顶已经改变为新工作目录了：

```
0 /usr
```

现在，使用 **pushd** 命令向栈中压入 3 个新目录名称。必须采用 3 条单独的命令进行压入，然后使用 **dirs** 显示栈的内容：

```
pushd /lib
pushd /var
pushd /etc
dirs -v
```

输出为：

```
0 /etc
1 /var
2 /lib
3 /usr
```

现在栈中包含 4 个目录名称。下面显示当前工作目录：

```
pwd
```

输出为:

```
/etc
```

注意我们没有显式地改变工作目录。每当栈顶元素(#0)发生改变时,工作目录也会自动发生改变。*

接下来,使用 **popd** 从栈顶弹出一个目录名称,然后显示栈的内容和当前工作目录:

```
popd
dirs -v
pwd
```

dirs 命令的输出为:

```
0 /var
1 /lib
2 /usr
```

pwd 命令的输出为:

```
/var
```

popd 命令将/etc 从栈中弹出,从而使/var 成为栈顶元素。发生这件事的同时,/var 成为当前工作目录。我们可以使用 **pwd** 命令确认这一点。

如果再看一下这些命令的语法,就会发现 **dirs** 有几个选项。如果没有选项,那么 **dirs** 将以压缩格式显示目录栈,即所有的名称都在一行上显示。如果目录名称涉及到 home 目录,那么 **dirs** 将以~(波浪号)字符表示 home 目录。带上-l(long, 长)选项以后,**dirs** 将显示 home 目录的完整名称。最后,加上-v(verbose, 详细)选项以后,**dirs** 将分行显示每个目录名称,且每行都有行号。下面体验一下,将 home 目录压入到栈中,然后试一试下述命令:

```
pushd ~
dirs
dirs -l
dirs -v
dirs -l -v
```

dirs 命令还有一个选项,但是这个选项与名称的显示没有关系。**-c**(clear, 清除)选项清空目录栈。当希望清空栈,从头开始时,可以使用这个选项。下面体验一下,先使用 **dirs -c**(清空目录栈)命令,然后再使用 **dirs -v**(显示目录栈)命令。在输入这些命令之前,看看您能不能回答这个问题:第二条命令会不会显示一个空栈?

```
dirs -c
dirs -v
```

* 您可能听说过魔术师 Harry Houdini(1874-1926)的传说。Houdini 经常表演神秘的心灵感应魔术,他不使用 **pwd** 命令就可以猜出他人的工作目录。那么秘诀是什么呢?在没有人看到时,Houdini 就使用 **dirs** 命令偷偷地看一下栈顶。

答案是永远都不会看到一个完全空的栈。这是因为栈顶是工作目录的名称，因为工作目录永远存在，所以目录栈中必须至少有一个名称。

此时，可以想象您正在想所有这些是多么的有趣(或者无聊，取决于个人的观点)，但是它们究竟有什么用处呢？我经常听到人们说，为什么要将目录名称每次一个地压入或弹出目录栈，从而来改变工作目录呢？为什么不使用 **cd** 呢？

如果您这样想，那么您是正确的。大多数时候，只有真正的极客才使用目录栈^{*}。实际上，如果只希望在两个目录间来回切换，则可以使用 **cd -**(本章前面描述)。所以，为什么还要学习更神秘的命令，并在栈上浪费大量的时间呢？

我讲授这些的原因在于目录栈在一个方面特别有用，即可以使用 **pushd** 命令跳转到栈的中间，将一个目录名称“压入”到栈顶。这样做时，也就改变了工作目录。

这听起来非常复杂，但事实上并没有这样复杂。它快速、容易，而且十分强大。下面举例说明它的工作方式。首先输入下述命令：

```
cd
dirs -c
pushd /lib
pushd /var
pushd /etc
dirs -v
```

cd 命令切换到 **home** 目录。**dirs -c** 命令清空目录栈。此时，除了工作目录(就是 **home** 目录，用 **~** 表示)之外，栈是空的。接下来的 3 条 **pushd** 命令在栈顶压入目录名称。最后的 **dirs** 命令显示目录栈的内容。该命令的输出为：

```
0 /etc
1 /var
2 /lib
3 ~
```

现在目录栈中有 4 个目录名称，其中工作目录 **/etc** 位于最顶部(#0)。假设您已经在 **/etc** 目录中工作了一段时间，希望切换到目录 **/lib**(#2)。只需输入 **pushd**，后面跟一个+(加号)和数字 2：

```
pushd +2
```

这将告诉 shell 将#2 目录(**/lib**)移动到栈顶(#0)。在 **/lib** 变为#0 时，它也将成为工作目录。最终实现的结果是从栈的中间选择#2 目录，并使它成为工作目录。此时，如果使用 **dirs -v** 显示目录栈的话，它看上去如下所示：

```
0 /lib
1 ~
2 /etc
3 /var
```

这是怎样发生的呢？当将目录压入到栈顶时，其上面的目录并不会丢失。它们只是在

^{*} 如果您碰巧就是大量使用目录栈的人，那么您应该知道我以最友善的意义使用单词“geek，极客”。

栈中向下移动。在这个例子中，当把目录#2 向上移动到栈顶时，目录#0 和#1 将依次转到栈的底部。

必须承认，这个例子是人为的。毕竟，输入命令创建一个目录栈、显示栈的内容，然后向栈顶压入目录名称，这样做毫无意义，因为只需键入一条简单的命令行就可以完成同样的事情：

```
cd /lib
```

但是，如果目录的名称很长，该怎么办呢？例如，假设您位于自己的 `home` 目录 `/home/harley` 中。您现在输入下述 `pushd` 命令，向目录栈中压入 4 个非常长的名称：

```
pushd /home/harley/source/current/calculate/source/2-1.05
pushd /usr/include/linux/nfsd
pushd /home/harley/archive/calculate/source/1-0.31
pushd /usr/share/dict/
```

如果准备在这些目录中完成大量的工作，那么不停地键入这些名称确实比较烦人。作为替代，您可以将它们一次性压入到栈中。然后，每当需要时，就可以向栈顶压入希望的目录名称。例如，假设您现在在 `dict` 目录中。过了一会，您希望切换到 `nfsd` 目录。首先显示栈的内容，查看希望压入的目录名称的行号：

```
dirs -l -v
```

输出为：

```
0 /usr/share/dict
1 /home/harley/archive/calculate/source/1-0.31
2 /usr/include/linux/nfsd
3 /home/harley/source/current/calculate/source/2-1.05
4 /home/harley
```

您需要做的就是将目录#2 压入栈顶：

```
pushd +2
```

然后，当需要改变到另一个目录时，只需再次显示目录栈的内容，将另一个名称压入到栈顶(每次都显示栈的内容非常重要，因为在压入目录时行号会发生变化)。

目录栈元素的删除和添加非常容易。要删除目录栈中的名称，可以使用 `popd` 命令，后面跟名称的行号。向栈中添加名称时，可以使用前面描述的 `pushd`。例如，为了将栈中的#2 名称移除，可以使用：

```
popd +2
```

为了向栈中压入 `/home/weedly/bin`，可以使用：

```
pushd /home/weedly/bin
```

在第 13 章中，当讨论历史列表时，我们示范了如何显示命令列表，然后通过事件号引用特定的命令。您能看出目录栈与此的相似之处吗？首先显示目录列表，然后通过号码

引用特定的目录(该相似性并不是偶然的)。

在结束本节之前,我们先示范一些非常酷的事情。为了使目录栈更易于使用,可以为 **dirs -v** 和 **pushd** 创建别名(参见第 13 章)。下述命令适用于 Bash(记住, Korn shell 不支持目录栈)。

```
alias d='dirs -v'
alias p=pushd
```

对于 C-Shell 家族,可以使用:

```
alias d 'dirs -v'
alias p pushd
```

一旦定义了这些别名,目录栈的使用就更简单了。为了显示目录栈,只需输入:

```
d
```

为了通过在栈顶压入一个新名称来改变工作目录,可以使用下述命令:

```
p /usr/lib
```

为了通过将一个已有的名称移到栈顶来改变工作目录,可以使用下述命令:

```
p +4
```

如果您现在有时间,则可以键入这些别名体验一下。在输入目录名称时,一定要使用自动补全(参见第 13 章)功能,从而使键入工作尽量少。如果希望别名永久化,可以将它们放入到环境文件(参见第 14 章)中。最后,本节中讨论的各个命令请参见图 24-3。

提示

如果打算重复使用相同的目录集,则可以在登录文件(参见第 14 章)中放入合适的 **pushd** 命令。通过这种方式,每次登录时,系统会自动地创建目录栈。

24.9 最重要的程序: ls

在全部 Unix 工具中,最重要的工具就是 **ls** 程序(发音为“L-S”),该程序用来显示有关目录内容的信息。为什么 **ls** 程序如此重要呢?为了回答这个问题,我们需要考虑 Unix 的本质特征。

正如第 6 章中所述,Unix 系统中的每个对象或者是文件,或者是进程。简单地讲,文件存放数据或者允许访问资源,进程是正在执行的程序。在您使用 Unix 时,我们称您为用户。但是,Unix 本身并不知道用户,而只知道用户标识(参见第 4 章)。实际上,在 Unix 系统中,只有用户标识才有真正的身份。因此,是用户标识而不是用户在进行登录、注销、拥有文件、运行程序、发送电子邮件等活动。

基于这一原因,每个 Unix 系统都有一个内部(inside)和一个外部(outside),并且内部和外部之间拥有明确的界限。内部包含所有的文件和进程,以及生活在虚拟环境中的用户标

识。外部就是您，即用户。内部和外部之间的界限由物理界面定义，例如：键盘、鼠标、显示器、扬声器等。尽管操作的大脑是您(用户)的，但是您不能进入 Unix 环境。因此，您无法直接感知 Unix 内部存在什么以及发生什么。

可以确定的是，您是主管，您的用户标识充当您的官方式代表。但是，当深入到 Unix 中时，您就只能依靠其他方面，就像飞行员在大雾中只能依靠仪表飞行一样。您看不到任何文件，任何进程。您甚至看不到自己的用户标识。您需要做的就是输入命令，然后解释输出。基于这一原因，最重要的工具就是那些充当您的耳朵和眼睛的工具，这些程序能够显示文件和进程的相关信息。也就是说，这些工具会帮助我们回答下述问题：“那里有什么？”和“发生了什么事情？”

在第 26 章中，将示范如何查看进程的状态(使用的主要工具是 **ps** 程序)。但是，尽管进程非常重要，大多数时间也只是让它们完成自己的工作。我们的大多数精力花费在思考和操作文件上。因为文件位于目录中，所以允许查看目录的工具特别重要，到目前为止，这类工具中最有用的就是 **ls**。

而这正是在 Unix 和 Linux 系统提供的数百个命令行程序中，**ls** 程序是最重要的那一个的原因。

24.10 列举目录内容：ls -CrR1

为了显示有关目录内容的信息，可以使用 **ls**(list files, 列举文件)程序。**ls** 是最经常使用的 Unix 程序之一。就其本身而言，它有许多控制输出的选项。例如，在我的一个 Linux 系统上，**ls** 就有 59 个选项(这并不是印刷错误)。非 Linux 系统拥有的选项较少，但是即便如此，选项通常也超过 30 个。

很明显，没有程序实际需要 30 个选项，更不用说 59 个选项。在接下来的讨论中，我们只讨论那些最重要的选项。更多的信息，可以查看联机手册(**man ls**)。在本节中，我们将介绍 **ls** 程序并讨论基本的选项。在接下来的几节中，将讨论 **ls** 程序更高级的特性，那时候将描述一些比较复杂的选项。

只考虑最重要的选项，**ls** 程序的语法为：

```
ls [-aCdFglrRs1] [name...]
```

其中 *name* 是目录或文件的名称。

在继续之前，请先花一点时间看一看各个选项，并注意该程序有一个 **-l**(小写字母“l”)选项和一个 **-1**(数字“1”)选项。这是两个不同的选项，不要混淆了。**-l**(字母 l)选项经常使用，而 **-1**(数字 1)极少使用。

ls 程序的默认行为就是按字母表顺序显示目录中各文件的名称。例如，为了列举 **/bin** 目录中的文件，可以使用：

```
ls /bin
```

如果希望查看不止一个目录的内容，则可以指定多个目录名称。例如，为了列举 **/bin**

和/etc 目录中的内容, 可以使用:

```
ls /bin /etc
```

如果不指定目录, 那么 **ls** 默认情况下将显示工作目录中的文件。因此, 为了查看工作目录中的内容, 可以使用:

```
ls
```

这个两个字母的单词是 Unix 中最经常使用的命令。

正如前面所讨论的, .(点号)字符是工作目录的缩写。因此, 下述两条命令等价:

```
ls
ls .
```

更有用的缩写是.., 该缩写代表的是父目录。因此, 为了列举工作目录的父目录中的内容, 可以使用:

```
ls ..
```

可以想见, 使用..多次可以在目录树中向上移动多层。例如, 为了列举工作目录的父目录的父目录中的内容, 可以使用:

```
ls ../..
```

当 **ls** 将结果发送给终端(通常是这种情况)时, 输出将以列的形式组织。列的数目会自动选择, 从而使名称可以适应屏幕或窗口。例如, 下面的内容就是/bin 目录的目录列表的前 7 行输出(在这个特定系统上, 实际输出有 20 行)。

awk	dmesg	kill	ping	stty
bash	echo	ksh	ps	su
cat	ed	ln	pwd	tcsh
chmod	egrep	login	rm	touch
cp	env	ls	rmdir	umount
cut	ex	mkdir	sed	uname
date	false	more	sort	vi
.
.
.

注意, 文件名以列的形式按字母顺序排列。也就是说, 要竖着阅读文件, 而不是横着读。正如第 23 章中解释的, /bin 目录中包含许多标准的 Unix 程序, 因此这个目录中的文件名看起来不会那么陌生。

当将 **ls** 的输出重定向到文件或者管道线时, **ls** 以每个文件名占一行的形式输出结果。这将使 **ls** 的输出易于为其他程序所处理(重定向和管道线在第 15 章中解释过)。一个常见的例子是:

```
ls | wc -l
```

wc -l 命令统计它接收的输入的行数。因此, 这个 **ls** 和 **wc** 命令的组合可以告诉您在工

作目录中有多少个文件。

如果由于某种原因,您希望强制 **ls** 以列的形式将输出写入到文件或管道线中,则可以使用 **-C** 选项(大写字母“C”),例如:

```
ls -C | less
```

如果希望强制 **ls** 程序以每个文件名占一行的形式将输出写到终端(而不是以列的形式),则可以使用 **-l** 选项(数字“1”):

```
ls -l
```

默认情况下, **ls** 按字母表顺序显示文件名(更准确地说, **ls** 按照区域设置的排序序列中字母的顺序进行排序。参见本章后面的讨论)。如果希望以相反的顺序显示文件名,则可以使用 **-r**(小写字母“r”)选项:

```
ls -r
```

我们在本节讨论的最后一个 **ls** 选项是 **-R**,它代表“recursive,递归”(稍后再加以解释)。该选项告诉 **ls** 程序列举指定目录中的所有直接或间接的子目录和文件的信息。换句话说, **ls -R** 显示整个目录树的信息。

例如,假设您希望查看系统中由用户创建的所有文件和子目录。这种情况下,需要显示 **/home** 目录的全部子孙:

```
ls -R /home
```

同样,为了列举工作目录的全部子孙,可以使用:

```
ls -R
```

这样的列表一般很长,所以通常将输出管道传送给 **less** 程序,从而在终端上每次显示一屏幕。因为输出传送给管道线,所以如果希望以列的形式输出则必须加上 **-C** 选项:

```
ls -CR /home | less
ls -CR | less
```

当希望使用 **-R** 选项时,记住还有一个 **-r**(reverse,相反)选项,因此一定要小心地键入。

名称含义

递归

在计算机科学中,递归的数据结构指由相同类型的小数据结构逐级构建而成的数据结构。目录树就是递归的,因为它们包含其他较小的目录树。

一些目录工具,例如 **ls**,提供有一个选项,可以处理整个目录树,也就是所有派生于特定目录的子目录和文件。因为认为这样的树是递归的,所以处理它们的选项通常命名为 **-r** 或者 **-R**。

24.11 排序序列、区域设置和 ls

在本章前面，提到 **ls** 的默认行为就是按字母表顺序显示目录中的文件名。这个说明看似直接，但是实际上并不如此。这是因为“字母表顺序”的定义在各个系统上并不相同。它取决于使用什么排序序列，而排序序列最终由区域设置定义。

正如第 19 章中讨论的，区域设置是一种技术规范，描述在与来自某一特定文化的用户沟通时所使用的语言 and 习惯。例如，您的区域设置可能设置为美国英语、英国英语、荷兰语、西班牙语等。就目前的学习而言，区域设置最重要的一个方面就是定义排序序列，即字符排列的顺序(参见第 19 章中的解释)。

系统的默认区域设置在系统安装时设置。如果使用的是美国英语，则区域设置或者被设置为基于 ASCII 码的 **C(POSIX)** 区域设置，或者被设置为 **en_US** 区域设置，后者是一种较新的国际体系的一部分。要查看自己的区域设置，可以使用 **locale** 命令。这将显示各种环境变量的值。我们希望查看的那一个变量是 **LC_COLLATE**，它指定的就是排序序列的名称，而正是排序序列决定着系统中“字母表顺序”的含义。

C 区域设置使用和 ASCII 码相同的排序序列。具体来说，所有的大写字母都分成一组，所有的小写字母都分成一组，并且大写字母在小写字母前面：**ABCDEF...abcdef...**。我们称这种排序序列为 **C** 排序序列，是因为 **C** 编程语言使用这种排序序列。

另一方面，**en_US** 区域设置使用字典排序序列，在这种排序序列中，大写字母和小写字母混杂在一起：**aAbBcCdDeEfF...**。

当使用 **ls** 列举文件时，文件显示的顺序取决于使用的排序序列。例如，假设您有 6 个文件，文件名分别为 **A**、**a**、**B**、**b**、**C** 和 **c**。如果使用的区域设置是 **C** 区域设置，那么在使用 **ls** 列举文件时，看到的将是：

```
A B C a b c
```

如果使用的是 **en_US** 区域设置，则看到的结果是：

```
a A b B c C
```

尽管这看上去似乎更接近于理想状况，但事实并不是这样。如果使用 **C** 区域设置，生活将会更加轻松。在本章后面讨论通配符时，您将会看到一个非常重要的与此相关的例子。在这里，我希望您做以下事情。

输入 **locale** 命令，这将显示定义区域设置的环境变量。如果 **LC_COLLATE** 变量被设置为 **C** 或者 **POSIX**，那么没有什么问题。如果它被设置为 **en_US**，那么我希望您将它永久地修改为 **C**(我在自己的系统上就是这样做的)。您所需做的就是在登录文件中添加合适的命令。如下所示，第一条命令适用于 Bourne Shell 家族(Bash、Korn Shell)，第二条命令适用于 C-Shell 家族(C-Shell、Tcsh)：

```
export LC_COLLATE=C
setenv LC_COLLATE C
```

有关区域设置和排序序列的详情请参见第 19 章，有关登录文件的讨论请参见第 14 章。

24.12 检查文件类型: ls -F

我们经常希望知道目录中所包含文件的类型。在这时, 您有 3 种选择: 使用带**-F** 选项的 **ls**、使用带**--color** 选项的 **ls**(只适用于 Linux)以及使用 **file** 命令。在接下来的 3 节中, 我们将分别讨论这些技术。

当使用带**-F(flag, 标志)**选项的 **ls** 时, **ls** 在特定类型文件的名称之后显示一个标志。这些标志如图 24-4 所示。其中最重要的标志是/(斜线), 它表示的文件类型是目录; 以及*(星号), 它表示的文件类型是可执行文件(例如程序或脚本)。在大多数情况下, 文件名称后面没有标志。这种情况表示文件是普通非执行文件。

标志	含义
空	普通文件: 非执行文件
*	普通文件: 可执行文件
/	目录
@	符号链接(参见第 25 章的讨论)
	命名管道/FIFO(参见第 25 章的讨论)

图 24-4 ls -F 命令显示的标志

例如, 假设您的工作目录中包含一个目录 **documents**、两个文本文件 **memo** 和 **essay**、一个程序(二进制文件)**spacewar** 以及一个命名管道 **tunnel**。为了在显示文件名称时显示文件的类型标志, 可以使用:

```
ls -F
```

输出为:

```
documents/ essay memo spacewar* tunnel|
```

24.13 检查文件类型: ls --color

如果使用的是Linux, 那么除**-F**选项外, 还有另外一种选择, 即通过使用**--color**选项来使用颜色指示各种不同类型的文件*(我们已经在第 10 章中讨论了以**--**开头的选项)。该命令的语法为:

```
ls --color[=always|=auto|=never] [name...]
```

其中 *name* 是目录或文件的名称。

当打开**--color** 时, **ls** 使用颜色来标示各种不同类型的文件。例如, 下述命令显示工作目录中文件的名称:

* 对于基于FreeBSD的系统来说, 包括OS X(Macintosh), 可以以相似的方式使用**-G**选项。

```
ls --color
```

当使用`--color`选项时,还有三种变体。第一种变体就是`--color=always`,这是默认设置。如果喜欢,还可以使用`yes`或`force`。因此,下述4条命令是等价的,它们都告诉`ls`使用颜色标示各种不同类型的文件:

```
ls --color
ls --color=always
ls --color=yes
ls --color=force
```

第二种变体是`--color=never`。这告诉`ls`不要使用颜色。如果基于某些原因,颜色选项打开了,而您又希望关闭它,则可以使用这种变体。如果喜欢,还可以使用`no`或`none`。因此,下述3条命令都告诉`ls`不要使用颜色:

```
ls --color=never
ls --color=no
ls --color=none
```

此时,您可能会奇怪,为什么有这么多的方式来进行实质上仅仅是“是”或“不是”的选择呢?答案就是添加颜色支持的程序员认为用户有可能指定`never`或`always`,也有可能指定`yes`或`no`。另外两个值`force`和`none`,添加它们是为了与其他版本的`ls`兼容。^{*}

通常,创建颜色的特殊代码混杂在输出中。当在显示器上显示输出时,这没有什么问题,但是当将输出发送给管道或者保存在文件中时,信息可能会有点乱。为了避免这样,可以通过将`--color`选项设置为`auto`来使用该选项的最后一种变体。这将告诉`ls`仅当输出要在终端上显示时才使用颜色。如果喜欢,还可以使用`tty`或者`if-tty`。因此,下述3条命令是等价的:

```
ls --color=auto
ls --color=tty
ls --color=if-tty
```

为了明白这一点,可以试一试下述两条命令。第一条命令强制使用颜色,因此在使用`less`查看时,所生成的特殊代码看上去就像无用信息。第二条命令检测到输出没有发送给终端,因此没有生成颜色代码,从而避免了`less`显示的问题。

```
ls --color=yes /bin | less
ls --color=auto /bin | less
```

同样,如果打开了颜色开关,那么在将输出保存到文件中时也希望将颜色关闭:

```
ls --color=auto > filelist
```

^{*} 在第2章中,我们解释过Linux是开放源代码软件,这意味着任何人都可以查看(甚至修改)其源代码。虽然`--color`选项的各种变体没有详细的文档资料,但是,通过阅读`ls`程序的源代码,我能够分析出它们之间的细微差别。

如果您确实希望理解一个程序的工作方式,而这个程序又没有详细的文档资料,那么请记住没有什么事情是神秘的。如果您有一点点的C知识,就可以阅读源代码。理解程序的要旨并不是那么困难,而且阅读他人的程序也是提高编程水平的最佳方式之一。

许多人喜欢每次使用 **ls** 时都显示颜色。实际上, 在一些系统中, 通常为 **ls** 自动创建一个 **--color** 选项打开的别名。如果您是 Linux 用户, 而在使用 **ls** 时总是看到颜色——即使没有指定 **--color** 选项, 那么极有可能是在不知情的情况下使用了别名。

为了检查是不是这种情况, 可以通过在命令前面键入一个 ****(反斜线)命令, 告诉 **shell** 忽略任何别名(参见第 13 章)。如果现在输出中没有颜色, 那么可以断定您之前使用的就是别名。

```
\ls
```

如果希望永久关闭颜色, 只需自己创建一个别名作为替代, 例如:

```
alias ls='ls --color=no'
```

```
alias ls 'ls --color=no'
```

第一个别名适用于 Bourne Shell 家族, 如 **Bash**、**Korn Shell**; 第二个别名适用于 **C-Shell** 家族, 如 **Tcsh**、**C-Shell**。为了使别名永久化, 可以将别名放在环境文件中(有关别名的帮助, 请参见第 13 章。有关环境文件的内容, 请阅读第 14 章)。

如果希望永久打开颜色, 则可以使用下述别名替代环境文件中的别名:

```
alias ls='ls --color=yes'
```

```
alias ls 'ls --color=yes'
```

就个人而言, 我倾向于使用标志, 而不是颜色, 因此我建议您同时使用 **-F** 和 **--color**:

```
alias ls='ls -F --color=yes'
```

```
alias ls 'ls -F --color=yes'
```

提示

当使用带 **--color** 选项的 **ls** 时, 将使用不同的颜色指示不同类型的文件。这些颜色在称为 **LS_COLORS** 的环境变量中设置。通过修改这个变量可以自己定制颜色。如果觉得这听起来有趣的话, 可以先阅读一下 **dircolors** 程序的帮助信息:

```
man dircolors
info dircolors
```

这里的思想就是使用 **dircolors** 生成一条按照自己希望的方式设置 **LS_COLORS** 的命令, 然后就可以在环境文件中使用这条命令。

24.14 检查文件类型: file

到目前为止, 已经讨论了两种查看目录中所包含文件的类型的方式。我们可以使用带 **-F** 选项的 **ls** 命令在文件名之后显示一个标志指示文件的类型, 也可以使用带 **--color** 选项的 **ls** 程序用颜色来指示不同类型的文件(或者同时使用两种方式)。一种更复杂的检查文件类

型的方式就是使用 **file** 命令，该命令可以描述几千种类型的文件。该命令的语法为：

```
file [name...]
```

其中 *name* 是文件或目录的名称。该命令有许多选项，但是我们不需要使用它们(如果您感到好奇的话，可以查看该命令的说明书页)。

file 的使用相当直接，只需简单地指定一个或多个文件或目录的名称即可，例如：

```
file /etc/passwd /bin / ~/elmo.c /bin/ls
```

下面是一些典型的输出：

```
/etc/passwd:      ASCII text
/bin:             directory
/:               directory
/home/harley/elmo.c: ASCII C program text
/bin/ls:          ELF 32-bit LSB executable, Intel 80386
                  version 1 (SYSV), for GNU/Linux 2.6.9,
                  dynamically linked (uses shared libs),
                  stripped
```

前 4 个文件的输出易于理解。第一个文件(口令文件)包含纯 ASCII 文本。第二个和第三个文件都是目录。第四个文件包含的是 C 源代码。最后一个文件是一个可执行程序。在这里，**file** 给出了大量的技术信息，这些信息对程序员和系统管理员都非常有用。为了满足您的兴趣，下面给出它们的含义。

ELF: 可执行和链接格式(Executable and Linking Format)，可执行文件的标准文件格式。

32-bit: 字长。

LSB: 采用最低有效字节(Least Significant Byte)词序编译，x86 处理器使用。

executable: 可执行文件。

Intel 80386: 编辑文件的处理器体系结构。

version 1(SYSV): 内部文件格式的版本。

GNU/Linux 2.6.9: 编译程序的操作系统和内核的版本。

dynamically linked(uses shared libs): 使用共享库，而不是静态链接。

stripped: 将符号表移除的可执行文件。这是由 **strip** 程序完成的，其目的是缩减可执行文件的大小。

在我们的例子中，有两个目录：**/bin** 和 **/**(根目录)。注意 **file** 给出了目录本身的信息。如果希望 **file** 分析目录中的文件，则需要指定它们的名称。为了指定一个目录中的全部文件，可以使用所谓的“通配符”。我们将在本章后面讨论通配符。现在，先举一个例子。下述命令使用 **file** 分析目录 **/etc** 中的全部文件。因为输出相当长，所以我们将输出管道传递给 **less** 以每次显示一屏：

```
file /etc/* | less
```

24.15 掌握磁盘空间使用情况: `ls -hs`、`du`、`df`、`quota`

Unix 中有 3 个程序可以用来查看文件使用磁盘空间的情况, 这 3 个程序是 `ls -s`、`du` 和 `quota`。本节将依次讨论它们。

第一个程序是带 `-s`(size, 大小)选项的 `ls` 程序。这将告诉 `ls` 在每个文件名前面以 KB 为单位列出文件的大小。如果指定的是目录名称, 那么 `ls` 还将显示整个目录的总大小。下面举例说明:

```
ls -s /bin
```

下面是这条命令的一些输出(实际输出有 21 行)。

```
total 8176
 4 awk      12 dmesg     16 kill     40 ping     48 stty
712 bash    28 echo     1156 ksh    88 ps       32 su
28 cat      60 ed       35 ln      28 pwd     352 tcsh
44 chmod    4 egrep     32 login    48 rm       48 touch
76 cp       24 env      100 ls      24 rmdir    72 umount
40 cut      4 ex        36 mkdir    60 sed      24 uname
54 date     24 false    36 more     64 sort     592 vi
.           .           .           .           .
.           .           .           .           .
.           .           .           .           .
```

在第一行上, 显示的是该目录中所有文件占用的总空间大小是 8176KB。其他行显示各个文件所需的空间大小。例如, `cat` 文件使用了 28KB。对于 Linux 来说, 还可以使用 `-h`(human-readable, 适合人类阅读)选项显示一个合适的单位, 例如:

```
ls -sh /bin/cat
```

输出为:

```
28K /bin/cat
```

下一个程序是 `du`(disk usage, 磁盘使用), 可以用来显示文件的大小。该程序的语法为:

```
du [-achs] [name...]
```

其中 `name` 是目录或文件的名称。

当指定一个或多个文件的名称时, `du` 将显示这些文件所使用的存储空间总量。下面举例说明, 这个例子显示口令文件(参见第 11 章)的大小:

```
du /etc/passwd
```

在大多数系统上, 输出将以 1KB 为单位。例如, 下述输出告诉您口令文件占用了 8KB 的磁盘空间:

```
8 /etc/passwd
```


为了在显示大小时显示单位，可以使用**-h**(human-readable, 适合人类阅读)选项：

```
du -h /etc/passwd
```

该命令将把输出变为：

```
8.0K /etc/passwd
```

您可能会奇怪，为什么口令文件——通常是一个很小的文件——却要使用 8KB 磁盘空间呢？毕竟，8KB 可以存放 8192(8×1024)个字符，这要比口令文件中的数据多许多。原因是这样的，对于该文件系统而言，磁盘空间分配的数据块大小为 8KB。因此，即使文件比较小，它也占用 8KB 的磁盘空间(参见本章后面有关分配单元的讨论)。

正如前面所述，大多数版本的 **du** 以 1KB 为单位显示输出。但是，也有一些系统使用 512 字节大小的单元(该单位大小记录在 **du** 的说明书页中)。例如，Solaris 就是这种情况。在这样的系统上，通常有一个 **-k** 选项来强制 **du** 使用 1KB 作为单位。例如，在 Solaris 系统上，可以使用下述命令中的一个以 1KB 为单位显示口令文件使用的磁盘空间：

```
du -k /etc/passwd
du -hk /etc/passwd
```

到目前为止，我们已经学会使用 **du** 显示给定文件所使用的磁盘空间。但是，大多数时候，**du** 被用来统计一个特定目录树中全部文件所使用的磁盘空间。如果不指定名称，那么 **du** 将假定是工作目录。例如，下述命令从工作目录开始，显示每个子目录、子-子目录、子-子-子目录等的名称。除了每个名称外，**du** 还显示每个目录中的文件所占用的总磁盘空间。在输出的最后，还显示一个总数(因为输出相当长，所以我将输出管道传送给 **less**，以每次显示一屏)。

```
du -h | less
```

为了查看您的全部文件使用了多少磁盘空间，可以指定您的 home 目录：

```
du -h ~ | less
```

下述命令显示 **/usr/bin** 以及 **/etc** 中的全部文件所使用的磁盘空间大小：

```
du -h /usr/bin /etc | less
```

如果使用 **-s**(sum, 总和)选项，那么 **du** 将只显示总和，从而缩减大量的输出。依我的观点来看，这是 **du** 最有用的使用方式。下面举两个例子。第一个例子显示用户的个人文件使用的总磁盘空间(从自己的 home 目录开始)：

```
du -hs ~
```

第二个例子对 **/usr/bin**、**/bin** 和 **/etc** 目录完成同样的事情。第一条命令适用于 Bourne Shell 家族(Bash、Korn Shell)，第二条命令适用于 C-Shell 家族(Tcsh、C-Shell)。

```
du -hs /usr/bin /bin /etc 2> /dev/null
(du -hs /usr/bin /bin /etc > /dev/tty) >& /dev/null
```

这些命令都有点复杂，因此我们花一些时间来讨论它们。注意，我们通过将标准错误重定向到垃圾桶，从而将其抛弃(参见第 15 章)。我们这样做是因为，在 **du** 处理目录时，它可能会发现有的子目录由于没有权限而无法阅读。每次发生这种事情，**du** 将显示一个错误消息。抛弃掉标准错误就可以防止显示这些消息。

重定向标准错误的准确方法取决于所使用的 shell。因此，Bourne Shell 家族使用一条命令，而 C-Shell 家族使用另一条命令(所有的细节都在第 15 章中解释过)。

接下来，**-c(count, 统计)**选项在输出末尾显示总量。该选项在与**-s**和**-h**选项组合使用时最有用，下面举例说明。同前面一样，第一条命令适用于 Bourne Shell 家族，第二条命令适用于 C-Shell 家族：

```
du -csh /usr/bin /bin /etc 2> /dev/null
(du -csh /usr/bin /bin /etc > /dev/tty) >& /dev/null
```

这个组合选项(**-csh**)特别容易记住，因为它碰巧和 C-Shell 程序的名称相同。

最后，如果使用**-a(all, 全部)**选项，**du** 将显示它所处理的每个目录和文件的大小。这可能生成一个非常长的列表，但是它可以详细地描述磁盘空间的使用情况。例如，为了显示您自己的文件在磁盘空间使用方面的全部信息，可以指定自己的 home 目录：

```
du -ah ~ | less
```

下一个磁盘存储空间统计程序是 **df**(该名称代表 disk free-space，即磁盘可用空间)。这个程序显示每个文件系统已经使用了多少磁盘空间，以及还有多少磁盘空间可用。**df** 程序有许多选项，但是通常不需要使用它们。唯一一个需要提及的选项就是**-h**，该选项以适合于人类阅读的方式显示输出，即使用 KB、MB 和 GB 等存储单位取代块大小。在自己的系统上试一试下述命令，看看自己喜欢哪条命令：

```
df
df -h
```

下面是 Linux 中第一条命令的典型输出。在这个例子中，根文件系统(/)包含系统中几乎全部的数据，但只使用了 9% 的存储空间。另一个较小的文件系统 **/boot**，已经使用了 16% 的空间。

Filesystem	1K-blocks	Used	Available	Use%	
/dev/hda1	36947496	312446	31915940	9%	/
/dev/hdd1	99043	14385	79544	16%	/boot
tmpfs	192816	0	192816	0%	/dev/shm

下面是第二条命令的输出，可以和第一条命令比较一下：

Filesystem	Size	Used	Available	Use%	
/dev/hda1	36G	3.0G	31G	9%	/
/dev/hdd1	97M	15M	78M	16%	/boot
tmpfs	189M	0	189M	0%	/dev/shm

最后一个磁盘存储空间统计程序是 **quota**。如果共享一个 Unix 或 Linux 系统，那么系统管理员极有可能对每个用户标识强加一个配额，规定每个用户标识只允许使用多大的磁

盘空间。如果达到了配额，那么除非删除一些文件，否则无法再使用任何磁盘文件空间。

如果系统设置有这样的配额，则可以使用 **quota** 程序检查使用及限制情况：

quota

为了显示更多的信息，可以使用 **-v(verbose, 详细)** 选项：

quota -v

注意：4 个程序 **ls -s**、**du**、**df** 和 **quota** 以不同的方式估计存储空间的使用情况，所以如果各个数字有些出入的话，不要感到惊讶。

提示

如果使用的是共享系统，那么记住您正在“共享”非常重要。您应该时不时地使用 **du** 查看自己使用了多少磁盘空间。如果有不需要的文件，特别是大文件，则应该考虑将它们移除。

不要将配额当成一种限制，应该认为这是自己作为一个好邻居应该做的。

24.16 文件有多大？块和分配单元：dumpe2fs

磁盘存储器的大小以 KB、MB 或 GB 计量。1KB(1K)等于 $1024(2^{10})$ 字节，1MB 等于 $1048576(2^{20})$ 字节，1GB 等于 $1073741824(2^{30})$ 字节。在文本文件中，一个字节可以存放一个字符。例如，100 个字符要求 100 字节的磁盘存储空间。

我们已经讨论了如何使用 **ls -s** 以及 **du** 命令显示文件所使用磁盘空间的大小。在继续之前，还需要理解一个重要的观点。文件所使用磁盘空间的数量不同于文件中数据的数量。下面解释具体原因。

在文件系统中，空间以固定大小的组块进行分配，我们将固定大小的组块称为块(block)，根据文件系统的不同，块的大小有 512 字节、1KB、2KB 或 4KB 等。为文件所分配的最小磁盘空间数量就是一个块。下面分析块大小为 1K(1024 字节)的文件系统。典型的 Linux 系统就采用这种设置。在文件系统中，一个仅包含 1 字节的文件也要占据一个完整的块。如果文件的大小大于一个块，那么它需要第二个块。因此，一个包含 1024 字节数据的文件需要一个块。一个包含 1025 字节数据的文件需要两个块。

问题：一个包含 1000000 字节数据的文件需要多少个块呢？

答案：假定块的大小为 1024 字节，那么 1000000 除以 1024 约等于 976.6。因此，1000000 字节大小的文件需要占用 977 个块(977 个块有 1000448 字节)。

到目前为止，我们已经讨论了文件系统中数据的组织方式。但是，当文件写入到磁盘或其他存储介质上时，会发生什么情况呢？出于效率方面的考虑，磁盘存储空间也以固定大小的组块分配，我们称之为分配单元(allocation unit)或簇(cluster)。分配单元的大小取决于文件系统和存储设备。例如，在我的一个 Linux 系统上，块大小为 1K。但是，磁盘分配单元为 8K。因此，一个只有一个字节的文件实际上要占用 8K 的磁盘空间。

问题：一个文件包含有 8500 字节数据。这个文件需要多少个块呢？这要占用多大的磁盘空间呢？

答案：该文件包含 $8500/1024=8.3\text{K}$ 字节的数据。假定块大小为 1K，则这个文件需要 9 个块。假定磁盘空间以 8K 的分配单元分配，则这个文件占用两个分配单元，或者 16K 的磁盘空间。

如何确定系统的块大小和分配单元大小呢？我们首先从分配单元入手，因为其方法比较简单。我们的策略就是创建一个非常小的文件，然后查看这个文件占用多少磁盘空间。这个空间就是分配单元的大小。

第一步是创建一个非常小的文件。可以使用下述命令：

```
cat > temp
X
^D
```

首先，输入 **cat** 命令(参见第 16 章)从键盘读取输入，并将输入重定向到文件 **temp**。注意：如果 **temp** 文件不存在，那么 shell 将会创建这个文件；如果 **temp** 文件已经存在，那么 shell 将替换这个文件。

接下来，键入只包含一个字符的一行内容。在这个例子中，我键入了字母“X”，然后按<Return>键。该数据将写入到文件 **temp** 中。

最后，按[^]**D**(Ctrl-D)发送 **eof** 信号，以指示数据已经结束(参见第 7 章)。现在就拥有了一个非常小的文本文件，该文件包含两个字符：一个“X”，后面跟一个换行字符。

输入 **ls -l** 命令(本章后面解释)显示该文件所包含数据的数量：

```
ls -l temp
```

输出为：

```
-rw-rw-r-- 1 harley staff 2 Aug 10 11:45 temp
```

文件大小就显示在日期的前面。可以看出，文件包含两个字节的的数据。下面使用 **du** 程序(本章前面讨论的)查看该文件占用了多少磁盘空间：

```
du -h temp
```

下面是输出：

```
8.0K temp
```

可以看出，在我们的例子中，样本文件占用了 8K 的存储空间，即便该文件只包含两个字节的的数据。因此，可以得出结论，该系统的分配单元是 8K。

最后，使用 **rm** 程序(参见第 25 章)移除临时文件：

```
rm temp
```

查看文件系统的块大小需要一定的技巧。尽管一些文件程序，例如 **df**，可以显示“块大小”，但是这并不是文件系统的正式块大小：它只是程序使用的一个方便的单位。查看块

大小的确切方法依赖于使用的操作系统。对于 Linux 来说, 可以使用 **dumpe2fs** 程序; 对于 Solaris 来说, 可以使用 **fstyp -v**; 而对于 FreeBSD 来说, 可以使用 **dumpfs**。作为示例, 我们将以 Linux 进行示范(如果您需要这些程序更详细的描述, 可以查看联机手册)。

正如前面所述, 文件系统中所有的数据都组织成块。其中有一个块, 称为超块 (superblock), 这是一个特殊的数据区, 存放与文件系统本身有关的关键信息。对于 Linux 来说, 可以使用 **dumpe2fs** 程序检查超块的内容。特别地, 依靠它还有可能显示文件系统所使用块的大小, 下面是具体步骤。

(1) 查找表示文件系统的特殊文件的名称, 例如 **/dev/hda1**。该步骤可以使用 **df** 命令(本章前面讨论过)完成。

(2) 要运行 **dumpe2fs**, 必须是超级用户。使用 **su** 命令改变到超级用户(参见第 6 章)。

(3) 输入 **dumpe2fs** 命令, 后面跟特殊文件的名称。该命令将显示超块中的大量数据。我们所需的数字就位于包含有字符串 “Block Size” 的那一行之上。因此, 我们只需运行 **dumpe2fs** 命令, 并在输出中 **grep** “Block Size”。下面举例说明:

```
dumpe2fs /dev/hda1 | grep "Block size"
```

注意: 如果 shell 查找不到 **dumpe2fs** 程序, 则必须指定完整的路径名。该程序位于 **/sbin** 中:

```
/sbin/dumpe2fs /dev/hda1 | grep "Block size"
```

下面是一些示例输出。在这个例子中, 可以看出文件系统的块大小是 1K(1024 字节):

```
Block size: 1024
```

如果希望查看超块的全部信息, 可以将 **dumpe2fs** 的输出管道传送给 **less**:

```
dumpe2fs /dev/hda1 | less
```

在完成之后, 使用 **exit** 命令注销超级用户(参见第 6 章)。

24.17 使用通配符进行通配

每当键入以文件名作为参数的命令时, 可以通过使用特定的元字符——通配符 (wildcard)——指定多个文件名。第 13 章中讲过, 元字符就是由 shell 解释时拥有特殊含义的字符。当在文件名中使用通配符时, 通配符就拥有特殊的含义。下面举例说明。

假设您希望列举当前工作目录中所有以字母 “h” 开头的文件, 则可以使用:

```
ls h*
```

在这个例子中, *****(星号)就是一个匹配 0 个或多个字符的元字符。

乍一看, 通配符与第 20 章中讨论的正则表达式元字符极其相似。实际上, 通配符更加简单一些。此外, 它们只有一个用途: 当键入一条命令时匹配一组文件名。图 24-5 列出了基本的通配符及其含义。在继续之前, 请花一些时间将该表与第 20 章中汇总的正则表达式作一比较。

符号	含义
<code>*</code>	匹配任何 0 个或多个字符构成的序列
<code>?</code>	匹配任何单个的字符
<code>[list]</code>	匹配 <i>list</i> 中的任何字符
<code>[^list]</code>	匹配不在 <i>list</i> 中的任何字符
<code>{string1 string2...}</code>	匹配其中一个指定的字符串

图 24-5 用来指定文件名的通配符

每当键入使用文件名作为参数的命令时，可以使用通配符匹配多个文件名。本表示范了各种不同的通配符以及它们的用途。注意：当键入路径名时，不能匹配/字符，该字符必须显式键入。

当使用通配符时，shell 将解释模式，并在运行命令之前以合适的文件名替换模式。例如，假设您输入了：

```
ls h*
```

那么 shell 先将 **h*** 替换为当前工作目录中所有以字母 **h** 开头的文件名，然后运行 **ls** 命令。例如，假设工作目录中包含下述 6 个文件：

```
a data-old data-new harley h1 h2 z
```

如果输入前述命令，那么 shell 将把这条命令变换为：

```
ls h1 h2 harley
```

当然，在同一条命令中可以使用不止一种模式：

```
ls h* data*
```

在我们的例子中，shell 将把该命令变换为：

```
ls h1 h2 harley data-old data-new
```

根据所使用的 shell 的类型不同，使用通配符指定文件的正式称呼也有所不同。在 Bash 中，称之为路径名扩展(pathname expansion)；在 Korn shell 中，称之为文件名生成(filename generation)；在 C-Shell 或 Tcsh 中，称之为文件名替换(filename substitution)。当 shell 执行实际替换时，称之为通配(globbering)。有时候，单词 glob 用作动词，例如：“除非设置 **noglob** 变量，否则 C-Shell 将自动进行通配。”

正如前面所述，当 shell 通配通配符时，在参数传递给程序之前，通配符已变成实际文件名。如果使用的模式不匹配任何文件，那么 shell 将显示一个适当的消息。例如，假设工作目录中包含前面列举的文件，如果输入下述命令：

```
ls v*
```

该命令将列举所有以字符“**v**”开头的文件，因为当前工作目录中没有这样的文件，所以 **ls** 显示一个错误消息：


```
ls: v*: No such file or directory
```

现在您应该已经理解了主要概念, 下面详细讨论每个通配符。最重要的通配符是*(星号), 它匹配 0 个或多个字符。*通配符匹配除/(斜线)字符之外的任何字符, 这是因为/用作路径名中的定界符(如果希望指定/, 则必须自己键入)。例如, 下述通配符格式匹配相应的模式:

Ha* 以“Ha”开头的文件名

Ha*y 以“Ha”开头并且以“y”结尾的文件名

Ha*l*y 以“Ha”开头, 中间包含一个“l”, 并且以“y”结尾的文件名

?(问号)通配符匹配任意单个的字符(除/之外), 例如:

d? 以“d”开头的两个字符的文件名

?? 任何两个字符的文件名

?*y 至少两个字符, 并且以“y”结束的文件名

通过使用[和](方括号)将字符列表括起来, 可以指定一串字符。这表示指定字符列表中的任一个字符。例如:

spacewar.[co] “spacewar.c”或者“spacewar.o”

[Hh]* 以“H”或者“h”开头的文件名

为了匹配不在列表中的字符, 需要在列表的开头加一个^(音调符号)。例如, 下述命令显示工作目录中所有不以字母“H”或“h”开头的文件的名称:

```
ls [^Hh]*
```

在方括号中, 可以使用-指定字符的范围。例如, 模式**[0-9]**匹配任何 0 至 9 的数字。使用字母范围时与此相同, 只是必须特别小心: 字母范围会根据区域设置使用的排序序列有所变化(参见本章前面的讨论)。考虑下述两个例子:

[a-z]* 以小写字母开头的文件名

[a-zA-Z]*[0-9] 以大写字母或小写字母开头, 并且以数字结尾的文件名

如果使用的是 C 排序序列(C 区域设置), 则字母的顺序为 **ABCDEF...abcdef...**。因此, 上述示例的结果与期望相同。但是, 如果使用的是字典排序序列(en_US 区域设置), 那么上述示范的结果将与期望结果不同, 因为字典排序序列的字母顺序为 **aAbBcCdDeEfF...zZ**。

更具体地讲, 对于 C 排序序列来说, **[a-z]**匹配任何小写字母, 这正是我们希望的。对于字典排序序列来说, **[a-z]**匹配除“Z”之外的任何小写或大写字母, 而这不是我们所希望的。同样, 对于 C 排序序列来说, **[A-Z]**匹配任何大写字母。对于字典排序序列来说, **[A-Z]**匹配除“a”之外的任何大写字母或小写字母, 而这也不是我们所希望的。

基于这一原因, 我强烈建议您使用 C 排序序列, 而不是字典排序序列。如果要使用 C 排序序列, 必须确保将 **LC_COLLATE** 环境变量设置为 **C**, 而不是 **en_US**(本章前面已经介绍过设置过程)。

如果基于某些原因, 您决定使用字典排序序列, 则不能将**[a-z]**或**[A-Z]**作为通配符使用。但是, 可以使用几个预定义的字符类。图 24-6 中列举了一些最重要的预定义字符类。有关预定义字符类的详细讨论, 请参见第 20 章。

类	含义	类似于
<code>[:lower:]</code>	小写字母	<code>[a-z]</code>
<code>[:upper:]</code>	大写字母	<code>[A-Z]</code>
<code>[:digit:]</code>	数字	<code>[0-9]</code>
<code>[:alnum:]</code>	大写和小写字母、数字	<code>[A-Za-z0-9]</code>
<code>[:alpha:]</code>	大写和小写字母	<code>[A-Za-z]</code>

图 24-6 通配符：预定义字符类

通配符可以使用范围匹配指定字符集中的字符。最常见的范围包括[\[a-z\]](#)匹配小写字母，[\[A-Z\]](#)匹配大写字母。正如正文中解释的，这些范围适用于 C 区域设置，而不适用于 [en-US](#) 区域设置。作为备选方法，可以使用预定义字符类替代范围，本表列举了一些最重要的预定义字符类。更多的信息请参见第 20 章。

下面举例说明。假设您希望显示最古老的 Unix 程序的名称。这些程序的名称大多数由 2 个小写字母构成，例如 `ls` 和 `rm`。查看此类程序最好的位置是 `/bin` 和 `/usr/bin`(参见第 23 章)。如果使用的是 C 区域设置，则可以使用下述命令。在自己的系统上试一试，看看会查找到什么。

```
ls /bin/[a-z][a-z] /usr/bin/[a-z][a-z]
```

如果使用的是 `en_US` 区域设置，那么在遇到两个字母的程序名称中包含有大写字母时，会出现奇怪的结果。对于这个情况来说，正确的命令应该为：

```
ls /bin/[:lower:][:lower:] /usr/bin/[:lower:][:lower:]
```

您现在应该明白我推荐总是使用 C 排序序列的原因。

到目前为止，已经讨论了文件名扩展中使用的 3 种不同类型的通配符：匹配 0 个或多个字符的 `*`、匹配任何单个字符的 `?` 以及定义字符集合的 `[]`。需要讨论的最后一个通配符模式允许指定不止一个字符串，然后依次匹配每个字符串。这样做时，需要使用 `{和}`(花括号)将一串由逗号分隔开的模式括起来。例如：

```
{harley,weedly}
```

重点：逗号前面和后面不能有空格。

当以这种方式使用花括号时，花括号告诉 shell 依次使用每种模式形成一个单独的文件名。我们称其为花括号扩展(brace expansion)。花括号扩展只适用于 Bash、Tcsh 和 C-Shell(不适用于 Korn Shell 和 FreeBSD Shell)。当处理命令时，花括号扩展在文件名扩展之前完成。

下面举例说明。假设您希望列举目录 `/home/harley`、`/home/weedly` 和 `/home/tln` 中所有文件的名称，则可以明确指定全部 3 个目录的名称：

```
ls /home/harley /home/weedly /home/tln
```

如果使用花括号，那么命令就比较简单：

```
ls /home/{harley,weedly,tln}
```

下面再举一个例子。假设您希望将文件 `olddata1`、`olddata2`、`olddata3`、`newdata1`、`newdata2` 和 `newdata3` 中的内容组合起来，并将输出存储在一个新文件 `master` 中。可以使用下述命令之一：

```
cat olddata1 olddata2 olddata3 newdata1 newdata2 newdata3 > master
cat {old,new}data{1,2,3} > master
cat {old,new}data[1-3] > master
```

(**cat** 程序用来组合文件, 参见第 16 章的讨论。> 字符重定向标准输出, 参见第 15 章的讨论。)

花括号扩展非常重要, 因为它可以以两种方式使用。第一种, 正如前面所示, 花括号可以匹配一组拥有共同名称的文件。第二种, 当创建新文件时, 花括号还可以用来描述不存在的文件名。例如, 假设您的 **home** 目录中包含一个子目录 **work**。下面两条 **mkdir** 命令都可以在 **work** 目录中创建 4 个新子目录。注意使用花括号是多么的方便:

```
mkdir ~/work/essays ~/work/photos ~/work/bin ~/work/music
mkdir ~/work/{essays,photos,bin,music}
```

最后一个例子, 在第 25 章中, 我们将学习如何使用 **touch** 命令快速地创建空文件。假设您希望创建下述 5 个新文件:

```
dataold datanew databackup datamaster datafinal
```

使用花括号扩展时, 命令如下所示:

```
touch data{old,new,backup,master,final}
```

名称含义

通配符、通配

术语“通配符”来源于扑克和其他纸牌游戏, 在这些游戏中, 通常指定一些特定的牌作为“万能牌”。在扑克牌中, 万能牌可以用作不同的值。

“通配”指使用通配符将模式扩展为一串文件名。术语“通配”可以追溯到最早期的 Unix shell, 甚至在 Bourne Shell(参见第 11 章)之前。那时, 通配符扩展由一个独立的程序(/etc/glob)执行, 该程序由 shell 调用。没有人知道将该程序命名为 **glob** 的原因, 因此您自己可以虚构这方面的理由。

在 Unix 社区中, 通配的思想非常普遍, 以至于极客们在每天的会谈中都使用通配的思想。例如, 假设一名极客向另一名极客发送下述文本消息: “您最喜欢哪一部 Star Trek(星际旅行)电影: ST:TOS、ST:TNG 还是 ST:DS9?” 第二名极客可能回答: “我不看 ST:*。”

类似地, 有时候可能看到 UN*X 用来表示任何类型的 Unix 或 Linux。这要追溯回 20 世纪 70 年代, 那时 AT&T 公司声称 UNIX 是注册商标, 任何人在没有授权的情况下不能使用它。例如, AT&T 公司律师就说 UNIX 是一个形容词, 而不是一个名词, 人们永远不能只引用“UNIX”, 而必须是“UNIX 操作系统”。对于这种愚蠢的最终反应就是许多 Unix 极客开始使用 UN*X 来指任何类型的 Unix。

24.18 点文件(隐藏文件): ls -a

默认情况下, **ls** 程序不显示以.(点号)字符开头的文件名。因此, 如果您使用一个文件, 而又不希望每次使用 **ls** 程序时都看到这个文件, 那么可以使这个文件的名称以一个点号开头。正如第 14 章中讨论的, 这类文件称为点文件或者隐藏文件。大多数时候, 点文件由存

放配置数据或初始化命令的程序使用。例如，所有的 shell 都使用点文件，vi/Vim 编辑器(参见图 24-7)也是如此。

为了显示隐藏文件的名称，在使用 ls 程序时需使用 -a(all files, 全部文件)选项。例如，为了查看您自己的隐藏文件的名称，可以切换到自己的 home 目录中，使用 la -a 命令：

```
cd
ls -a
```

您很可能还会看到一些以点开头的目录名称。这样的目录也是隐藏的，除非使用 -a 选项，否则也看不到它们。

当使用 -a 选项时，将看到所有的文件。然而，没有选项来仅显示点文件。不过，通过使用通配符，就可以限制文件名列表，使其只显示点文件。例如，下述命令就显示工作目录中所有以.开头，后面是字母的名称：

```
ls .[a-zA-Z]*
```

下述命令稍微有点复杂，但是更加有用。该命令同样显示点文件。但是，它省略了.和..以及隐藏目录的内容：

```
ls -d .??*
```

图 24-7 列举了我们已经讨论过的标准点文件的名称(第 14 章讨论过 shell，第 22 章讨论过 vi 和 Vim)。这些文件可能就是某一天当您希望改变时需要使用的文件。在自己的 home 目录中，极有可能发现许多其他的点文件。除非知道自己正在做什么，否则不要管它们。

文件名	应用
.bash_login	登录文件：Bash
.bash_logout	注销文件：Bash
.bash_profile	登录文件：Bash
.bashrc	环境文件：Bash
.cshrc	环境文件：C-Shell、Tcsh
.exrc	初始化文件：vi、Vim
.history	历史文件：Bash、Korn Shell、C-Shell、Tcsh
.login	登录文件：C-Shell、Tcsh
.logout	注销文件：C-Shell、Tcsh
.profile	登录文件：Bash、Korn Shell、Bourne Shell
.tcshrc	环境文件：Tcsh
.vimrc	初始化文件：Vim

图 24-7 shell 和 vi/Vim 使用的点文件

任何以.(点号)开头的文件名都称为点文件或隐藏文件。除非使用 -a 选项，否则 ls 程序不显示点文件的名称。

点文件最常见的应用就是程序使用它来存放配置数据或初始化命令。例如，这里就示范了一些由各种不同的 shell 和 vi/Vim 文本编辑器使用的点文件。

提示

home 目录中的大多数点文件都非常重要。在编辑这些文件之前,最好是先做一个备份。在做备份时,可以使用 **cp** 程序(参见第 5 章),例如:

```
cp .bash_profile .bash_profile.bak
```

如果以后不小心破坏了文件,就能够恢复这个文件。这样做时,可以使用 **mv** 命令(参见第 25 章),例如:

```
mv .bash_profile.bak .bash_profile
```

24.19 长目录列表: **ls -dhl tu**

当使用 **ls** 程序时,有几个选项可以用来和文件名一起显示许多信息。其中最有用的选项是 **-l** 选项,它代表“long listing(长列表)”:

```
ls -l
```

如果列表过长,超出了屏幕的显示范围,则可以将列表管道传送给 **less**:

```
ls -l | less
```

下面示范一些输出,然后对其进行分析:

```
total 32
-rw-rw-r-- 1 harley staff 2255 Apr 2 21:52 application
drwxrwxr-x 2 harley staff 4096 Oct 5 11:40 bin
drwxrwxr-x 2 harley staff 4096 Oct 5 11:41 music
-rw-rw-r-- 1 harley staff 663 Sep 26 20:03 partylist
```

最右边是 4 个文件名: **application**、**bin**、**music** 和 **partylist**。在最左边,每一行的开头有一个单字母的指示符显示文件的类型。我们稍后再讨论文件的类型。现在,只需说明:表示常规文件, **d** 表示目录。因此, **application** 和 **partylist** 是常规文件,而 **bin** 和 **music** 是目录。

文件名的左边是时间和日期。这称为修改时间(modification time)。它显示文件的最后修改时间。在我们的例子中,文件 **application** 在 4 月 2 日下午 9:52 最后一次修改(记住,Unix 使用 24 小时制时钟,参见第 8 章和附录 F)。

作为选择,还可以在使用 **-l** 选项时使用 **-u** 选项,显示文件的访问时间(access time),而不是文件的修改时间。访问时间显示上一次读取文件的时间。例如:

```
ls -lu application
```

该命令的输出如下所示。从前述输出中可以看出,文件 **application** 的最后修改时间是 4 月 2 日下午 9:52。但是,从下述输出中可以看出,该文件的最后读取时间是 4 月 11 日下午 3:45:

```
-rw-rw-r-- 1 harley staff 2255 Apr 11 15:45 application
```

如果希望按时间顺序显示文件,则可以使用 **-t** 选项:

```
ls -lt
ls -ltu
```

下面是第一条命令的输出，按从最新(最近修改的)到最旧(非最近修改的)的顺序显示：

```
total 32
drwxrwxr-x 2 harley staff 4096 Oct  5 11:41 music
drwxrwxr-x 2 harley staff 4096 Oct  5 11:40 bin
-rw-rw-r-- 1 harley staff  663 Sep 26 20:03 partylist
-rw-rw-r-- 1 harley staff 2255 Apr  2 21:52 application
```

如果组合使用-t 选项和-r(reverse, 相反)选项, 那么 ls 将按照从最旧到最新的顺序显示文件:

```
ls -lrt
ls -lrtu
```

例如:

```
total 32
-rw-rw-r-- 1 harley staff 2255 Apr  2 21:52 application
-rw-rw-r-- 1 harley staff  663 Sep 26 20:03 partylist
drwxrwxr-x 2 harley staff 4096 Oct  5 11:40 bin
drwxrwxr-x 2 harley staff 4096 Oct  5 11:41 music
```

提示

假设工作目录中有大量的文件, 而您希望显示最近修改的文件的信息。那么最简单的方法就是根据时间以相反的顺序显示文件:

```
ls -lrt
```

为了显示最近访问过的文件, 可以使用:

```
ls -lrtu
```

因为您处理的是一个目录, 所以大多数文件名将滚动出屏幕的范围。但是, 这没有关系, 因为您关心的是最后几行。

在列表的最顶端, ls 显示所有被列举文件所使用的文件系统块的总数量。在本例中, 两个文件和两个目录共使用了 32 个文件系统块(有关文件系统块的解释, 请参见本章前面的讨论)。

在日期的左边, 是以字节为单位的文件大小。如果文件是文本文件, 那么每个字节存放一个字符的数据。例如, 文件 **partylist** 就是一个文本文件, 包含 663 个字符, 包括文本每行末尾的新行字符。同样, 文件 **application** 包含 2255 字节的数据, 也包括新行字符。重要的是要意识到这里看到的数字显示了文件中包含的实际数据量, 而不是文件占用的存储空间。如果希望查看文件占用的存储空间, 则必须使用 **du** 或者 **ls -s** 命令, 这两条命令在本章前面讨论过。

默认情况下, ls 以字节为单位显示文件的大小, 这样, 当文件非常大时容易混乱。为了以千字节(KB)或兆字节(MB)为单位显示文件的大小, 可以使用 **-h(human-readable, 适合**

人类阅读)选项:

```
ls -hl
```

例如, 下述输出显示与上例相同的文件:

```
total 32
-rw-rw-r-- 1 harley staff 2.3K Apr  2 21:52 application
drwxrwxr-x 2 harley staff 4.0K Oct  5 11:40 bin
drwxrwxr-x 2 harley staff 4.0K Oct  5 11:41 music
-rw-rw-r-- 1 harley staff 663 Sep 26 20:03 partylist
```

注意两个目录, 每个目录使用 4096 字节, 正好是 4K。这是因为这一特定的系统使用 4K 大小的分配单元, 而且每个目录最少使用 1 个分配单元(分配单元在本章前面讨论过)。重要的是记住数量 4K 指的是目录本身的大小, 而不是目录内容的大小。尽管我们在谈论目录时, 就好像它“包含”大量的文件, 其实这只是一个隐喻。目录只占用少量的存储空间, 因为它们包含的是有关文件的信息, 而不是文件本身。

在每行的最左边, 第一个字符显示的是文件的类型。这有若干种可能性, 如图 24-8 所示。正如前面所述, 最重要的字符是表示普通文件的 **-** 和表示目录的 **d**。尽管 **-** 字符标识普通文件, 但是它并不知道文件的任何内容。如果希望了解更多的信息, 可以使用 **file** 命令(本章前面描述)。例如, 从上述输出中可以看出, **partylist** 就是一个普通文件。如果希望了解更多的信息, 可以输入:

```
file partylist
```

输出为:

```
partylist: ASCII text
```

下面继续讨论文件指示符, 较不常见的字符有表示符号链接(参见第 25 章)的 **l**(小写字母“l”)、代表命名管道(参见第 23 章)的 **p** 以及代表特殊文件(参见第 23 章)的 **b** 和 **c**。当涉及到特殊文件时, Unix 将区分两种类型的设备。每次处理一个字节数据的设备(如终端)称为字符设备。每次处理固定数量字节数据的设备(如磁盘)称为块设备。字母 **c** 标识表示字符设备的特殊文件, 字母 **b** 标识表示块设备的特殊文件。

指示符	含义
-	普通文件
d	目录
l	符号连接
b	特殊文件(块设备)
c	特殊文件(字符设备)
p	命名管道/FIFO

图 24-8 ls -l 使用的文件类型指示符

当对 **ls** 使用 **-l**(long listing, 长列表)选项时, 每个文件的信息分别显示在一行上。在每行的最左边, **ls** 显示一个单独的字符指示文件的类型。这里列举的是最重要的文件类型指示符。

在文件大小的左边是两个名称：文件属主的用户标识和组，以及该用户标识所属的组。在我们的例子中，所有的文件由用户标识 **harley** 拥有，而 **harley** 位于组 **staff** 中(除非使用了 **-g** 选项，否则一些版本的 Unix 不显示组)。用户标识的左边是一个显示该文件有多少个链接的数字。最后，再左边是 9 个字符的字符串(第一个字符的右边)显示文件的权限。我们将在第 25 章中讨论这 4 个概念：文件属主、组、链接和权限，那时候将更加详细地讨论 **ls -l** 程序的输出。

当指定目录的名称时，**ls** 列举该目录中文件的信息。例如，为了以长列表显示目录 **/bin** 中的所有文件，可以使用：

```
ls -l /bin | less
```

如果希望显示目录本身的信息，则可以使用 **-d(directory, 目录)** 选项。这将告诉 **ls** 程序将目录按文件进行显示。例如，为了显示目录 **/bin** 本身的信息，而不是目录 **/bin** 的内容，可以使用：

```
ls -dl /bin
```

下面是一些样本输出：

```
drwxr-xr-x 2 root root 4096 Dec 21 2008 /bin
```

这是一个便利的选项，大家需要记住，当列举大量的文件时，有一些文件是目录，而 **ls** 程序将显示每个目录内部不必要的信息。当使用 **-d** 选项时，它将告诉 **ls** 不要查看目录的内部。

-l 选项显示的信息可以通过将输出管道传送给过滤器(参见第 16~19 章)以众多富有想象力的方式使用。下面举两个例子来说明这一思想。为了列举 9 月份最后一次修改的所有文件的名称，可以使用：

```
ls -l | grep Sep
```

为了统计 9 月份最后修改的文件的数量，可以使用：

```
ls -l | grep Sep | wc -l
```

24.20 ls 使用过程中的有用别名

ls 程序的使用非常频繁。实际上，正如前面所述，我认为 **ls** 程序是整个 Unix 工具箱中最有用的程序。基于这一原因，用户通常会为 **ls** 定义别名，这样就能够方便地使用 **ls** 程序最经常使用的选项。一旦定义了自己喜欢的别名，就可以将这些别名放置在环境文件中使别名永久化(有关别名的详细讨论，请参见第 13 章；有关环境文件的详细讨论，请参见第 14 章)。

ls 可以使用两种类型的别名。第一种是重新定义 **ls** 本身的别名。例如，假设无论何时，当使用 **ls** 程序时，总是希望使用 **-F** 和 **--color** 选项，则只需使用下述别名之一即可。其中第

一个别名适用于 Bourne shell 家族，第二个别名适用于 C-Shell 家族：

```
alias ls='ls -F --color=auto'
alias ls 'ls -F --color=auto'
```

第二种类型的别名为 **ls** 的特定变体生成一个全新的名称。下面是适用于 Bourne Shell 家族的别名：

```
alias ll='ls -l'
alias la='ls -a'
alias lla='ls -la'
alias ldot='ls -d .??*'
```

对于 C-Shell 家族来说，应当使用下述别名：

```
alias ll 'ls -l'
alias la 'ls -a'
alias lla 'ls -la'
alias ldot 'ls -d .??*'
```

这些别名可以方便地显示长列表(**ll**)、全部文件列表(**la**)、全部文件的长列表(**lla**)以及仅显示点文件的列表(**ldot**)。例如，一旦定义了 **ll** 别名，就可以通过使用下述命令显示 **/bin** 目录的长列表：

```
ll /bin
```

为了显示工作目录中文件的长列表，包括点文件，可以使用：

```
lla
```

为了只显示点文件，可以使用：

```
ldot
```

我的建议就是将这些别名放置在环境文件中，并花一些时间练习使用它们。一旦习惯了使用这些别名，那么您就离不开它们了。

24.21 显示目录树：tree

Linux 中有一个功能强大的工具 **tree**，可以绘制文件系统任何部分的图形。该程序的语法为：

```
tree [-adfFilrst] [-L level] [directory...]
```

其中 *level* 是树的深度，而 *directory* 是目录的名称。

为了查看该程序的工作方式，可以列举整个文件系统的树。因为该树特别大，所以需要将输出管道传送给 **less**。当不想继续阅读时，可以按 **q** 键退出。

```
tree / | less
```

大多数时候，可以使用 **tree** 程序可视化自己的那些文件。为了显示自己的那一部分文件系统，可以使用：

```
tree ~ | less
```

下面是一些典型的输出：

```
/home/harley
|-- bin
|   |-- funky
|   |-- spacewar
|-- essays
|   |-- history
|   |-- crusades
|   |-- renaissance
|-- literature
|   |-- kafka
|   |-- tolstoy
```

在这个例子中，home 目录中有两个子目录：**bin** 和 **essays**。**bin** 目录中包含两个文件。**essays** 目录中又包含两个子目录 **history** 和 **literature**，这两个子目录中都包含两个文件。

tree 程序有许多选项。我们将解释最重要的选项，以便大家练习使用。首先解释一些与 **ls** 相同的选项。**-a** 选项显示所有文件，包括点文件；**-s** 选项显示文件名的同时显示文件的大小；**-F** 选项显示一个标识文件类型的标志；**-r** 选项按相反顺序对输出排序；**-t** 选项按修改时间对输出排序。

另外，**tree** 还有自己的选项。其中最有用的选项就是 **-d**，该选项只显示目录，例如：

```
tree -d ~ | less
```

对于上述树结构来说，该命令的输出为：

```
/home/harley
|-- bin
|-- essays
|   |-- history
|   |-- literature
4 directories
```

-f 选项显示完整的路径名，例如：

```
tree -df ~ | less
```

对于上述树结构来说，输出为：

```
/home/harley
|-- /home/harley/bin
|-- /home/harley/essays
|   |-- /home/harley/history
|   |-- /home/harley/literature
4 directories
```

-i 选项省略缩进。当希望收集一组路径名时，这个选项非常有用：

```
tree -dfi ~ | less
```

对于上述树结构来说，输出为：

```
/home/harley
/home/harley/bin
/home/harley/essays
/home/harley/history
/home/harley/literature
4 directories
```

为了限制树的深度，可以使用 **-L**(limit, 限制)选项，后面跟一个数字。这将告诉 **tree** 只显示指定深度的树，例如：

```
tree -d -L 2 /home
```

最后一个选项是 **-l**，该选项告诉 **tree** 跟随所有的符号连接(参见第 25 章)，就像它们是真正的目录一样。

为了结束本节的讨论，下面举一个例子，说明如何使用 **tree** 查看整个文件系统中所有命名为 **bin** 的目录。这里的思想就是从 **root** 目录(/)开始，限制只搜索目录(**-d**)，显示完整的路径名(**-f**)，省略缩进(**-i**)，然后将输出发送给 **grep**(参见第 19 章)，并只选择那些以 **/bin** 结尾的行。命令为：

```
tree -dif / | grep '/bin$'
```

下面是一些示例输出：

```
/bin
/home/harley/bin
/usr/bin
/usr/local/bin
/usr/X11R6/bin
```

24.22 文件管理器

在本章中，已经讨论了目录的基本操作：创建、删除、移动以及重命名。我们还讨论了如何改变工作目录以及如何使用 **ls** 程序以各种不同的方式显示目录的内容。在第 25 章中，将讨论一些与普通文件相关的相似话题。特别地，我们将示范如何创建、复制、重命名、移动和删除文件，以及如何使用 **ls** 显示文件的信息。

在这两章中，使用了在 shell 提示处输入的基于文本的命令，即我们最初在第 6 章中讨论的标准的 Unix CLI(command-line interface, 命令行界面)。因为目录和文件命令的功能如此之大，所以我还希望大家知道另外一种方法。这种方法不用键入命令，而是使用文件管理器，一个用来帮助管理目录和文件的程序。

文件管理器使用整个屏幕或窗口来显示文件和目录的列表。通过按各种不同的键，可以快速方便地执行任何命令操作。每个文件管理器都有自己的工作方式，所以在此我们不会深入介绍细节，大家可以自学。一般情况下，掌握一个文件管理器的使用并不困难，但是一旦掌握了文件管理器的使用方法，那么您就会非常自然地使用它。开始之前，可以阅读文件管理器的内置帮助信息。

经典的文件管理器是 Norton Commander，这是一个相当流行的工具，最初由程序员 John Socha 于 1986 年为古老的 DOS 操作系统编写。多年以来，该工具一直延续着由 Socha 开发的双面板设计，并扩展了许多次。如果您是一名 Windows 用户，那么您可能已经体验过不同类型的设计，例如 Windows 的默认文件管理器 Windows Explorer。

通常，可以将文件管理器分成两大家族：基于 GUI 的文件管理器和基于文本的文件管理器。基于 GUI 的文件管理器设计用来在图形桌面环境中使用，例如 Gnome 或 KDE(参见第 5 章)。大多数桌面环境都提供有一个默认的文件管理器：对于 Gnome 来说是 Nautilus；对于 KDE 来说是 Konqueror(参见图 24-9)。但是，如果希望拥有更多的选择的话，也可以找到许多其他可用的免费图形文件管理器。基于文本的文件管理器在基于文本的环境中使用，例如，当使用虚拟控制台(参见第 6 章)或者使用终端仿真器(参见第 3 章)访问远程 Unix 主机时。

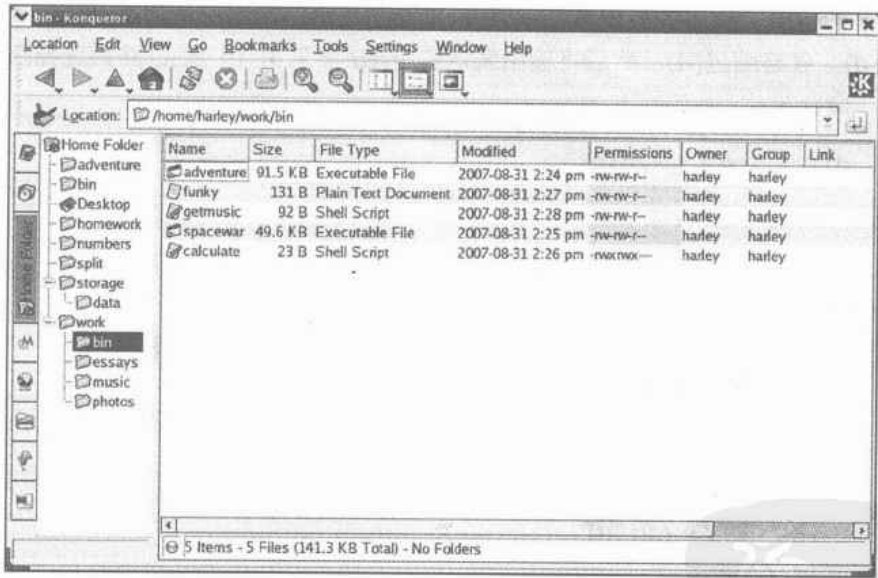


图 24-9 文件管理器示例

文件管理器是一个帮助管理文件和目录的程序。使用文件管理器管理文件和目录是在命令行上键入目录和文件命令的一种备选方法。这是 Konqueror 文件管理器的一张屏幕截图，Konqueror 是 KDE 桌面环境的默认文件管理器。

出于参考目的，下面给出一系列您可能希望尝试的文件管理器。

图形文件管理器：

- File Runner

- Gentoo
- Nautilus(Gnome 提供)
- Konqueror(KDE 提供)
- XFE [X File Explorer]

基于文本的文件管理器:

- FDclone(日本版的 FD, 一种 DOS 文件管理器)
- Midnight Commander(经典 Norton Commander 的兼容产品)
- Vifm(基于 vi 命令的文件管理器)

最后, 我还希望提及一个工具。您可以使用 Vim 文本编辑器(参见第 22 章)执行文件操作。使用目录名称启动 Vim, 然后就可以对该目录中的文件执行基本的操作。有机会时, 大家可以试一试。注意: 当以兼容模式(也就是使用 -C 选项)启动 Vim 时, 这一特性不管用。

24.23 练习

1. 复习题

1. 什么是路径名? 绝对路径名和相对路径名之间有什么区别?
2. 什么是工作目录? 工作目录还叫什么名字? 哪条命令显示工作目录的名称? 您能想出显示相同信息的另一种方式吗? 那些命令改变工作目录(不止一条)?
假定您希望一直提醒工作目录的名称, 那么如何进行安排呢? 提示: 考虑 shell 提示。
3. 使用什么程序来列举目录中的文件? 使用哪些选项来显示下述内容?
 - 全部文件, 包括点文件(隐藏文件)
 - 按字母表的相反顺序显示文件名
 - 整个树, 包括所有的子目录
 - 在每个文件名后显示标志(/ = 目录)
 - 以适合人类阅读的单位显示每个文件的大小
 - 长列表(权限、属主等)
 - 目录本身而不是目录内容的信息
 当查看目录列表时, 条目.和..各指什么?
4. 什么是通配? 什么是通配符? 5 种不同的通配符各是什么? 每种通配符各匹配什么内容? 通配和正则表达式有什么区别?
5. 使用什么程序来绘制目录树的图形? 该程序最有用的一个选项是什么选项?

2. 应用题

1. 从 home 目录开始, 使用尽可能少的命令, 创建下述目录:

temp/

```
temp/books/
temp/books/unix/harley
temp/books/literature/
temp/books/literature/english/
temp/books/literature/english/shakespeare
temp/books/literature/english/shakespeare/hamlet
temp/books/literature/english/shakespeare/macbeth
```

从 **temp** 目录开始，显示目录树的图形(只显示目录)。注意：如果使用的不是 Linux 的话，则可能找不到完成该任务的程序。

2. 使用上一问题中的目录结构，使用尽可能短的命令，创建下述空文件。提示：使用 **touch** 命令创建这些文件。**touch** 命令在第 25 章中解释，本章已经示范了 **touch** 命令的一个例子。

在 **harley** 目录中：创建 **notes**、**questions**、**answers**。

在 **english** 目录中：创建 **essay**、**exam-schedule**。

在 **hamlet** 目录中：创建 **quotes**。

接下来，从 **temp** 目录开始显示目录树的图形(显示所有的文件和目录)。

3. 清空目录栈。现在向栈中压入下述两个目录：

```
~/temp/books/unix/harley
~/temp/books/literature/english/
```

显示栈的内容，每个元素位于一行，并且显示行号。通过使用该目录栈，切换到目录 **harley**。

4. 创建两个版本的命令，显示工作目录中所有文件名中包含字符 “**backup**”，后面跟一个字符，再后跟两个数字的文件的访问时间。第一条命令应该使用预定义字符类。第二条命令应该使用范围。

3. 思考题

1. 您正在讲授 Unix 课程，现在该解释如何创建目录以及如何看待工作目录了。您有两种选择。第一种是抽象地解释概念：我们根据需要创建目录，然后使用 **cd** 命令改变工作目录。另一种方法以树的隐喻来介绍目录：目录就像树的树枝一样，使用 **cd** 就像从一个树枝移动到另一个树枝。您认为哪种方法比较好呢？为什么？

2. 为了显示文件的类型，可以使用 **ls -F** 或者 **file** 命令。为什么有必要拥有两种这样的命令？

文件操作

本章是有关 Unix 文件讨论的 3 章中的最后一章。在第 23 章中，详细讨论了 Unix 文件系统。当时，我们解释到 Unix 中共有 3 种类型的文件：目录、普通文件和伪文件。对于日常的工作，目录和普通文件最重要，因此，我希望能确保您掌握与这两种类型的文件相关的所有基本技能。在第 24 章中，我们讨论了如何使用目录。在本章中，将讨论普通文件操作的细节。

本章自始至终，当使用术语“文件”时，指的都是普通文件。因此，精确地说，本章的标题实际上应该是“普通文件操作”。

本章的规划如下：首先，我们将示范如何创建、复制、移动以及重命名普通文件。然后将讨论权限，即允许用户共享文件的属性。从那时起，我们将讨论深层次的内容，大家将明白文件的管理实际上包含着“链接”的处理。最后，我们将解释如何搜索文件以及如何处理搜索中查找到的文件。听起来这好像有许多内容，实际上也是如此，但是我保证，在阅读完本章之后，您就会发现物有所值。

25.1 创建文件：touch

如何创建一个文件呢？实际上，您不用创建文件。在需要时，Unix 会为您创建文件，您极少需要自己创建新文件。

有 3 种情形将自动地创建文件。第一种，当需要时，许多程序将自动地创建文件。例如，假设您使用下述命令启动 vi 编辑器(参见第 22 章)：

```
vi essay
```

该命令明确说明您希望编辑一个 **essay** 文件。如果 **essay** 不存在，那么 **vi** 将在第一次保存工作时创建该文件。在这个例子中，我使用了 **vi**，但是许多其他程序也同样适用该原则。

第二种，当将输出重定向到文件(参见第 15 章)时，如果文件不存在，那么 shell 将创建该文件。例如，假设您希望将 **ls** 命令的输出保存到文件 **listing** 中。您输入 **ls** 命令，并重定向输出：

```
ls > listing
```

如果 **listing** 不存在，那么 **shell** 将创建该文件。

最后一种，当复制文件时，复制程序将创建新文件。例如，假设您希望将文件 **data** 复制到文件 **extra** 中，输入下述命令：

```
cp data extra
```

如果文件 **extra** 不存在，则自动创建这个文件(**cp** 命令在本章后面解释)。

但是，假设基于某些原因，您希望创建一个全新的空文件。那么创建这种文件的最简单方法是什么呢？在第 24 章中，我们介绍过如何使用 **mkdir** 命令创建一个新目录。有没有相似的命令来创建一个普通文件呢？答案是没有，但是，有一条命令的副作用可以创建空文件。该命令称为 **touch**，下面解释这条命令的工作方式。

在第 24 章中，我们介绍过如何显示文件的修改时间(**ls -l**)或者访问时间(**ls -lu**)。修改时间是文件上一次改变的时间；访问时间是文件上一次读取的时间。**touch** 的主要目的就是不改变文件的情况下改变文件的修改时间和访问时间，就如同您伸出手轻轻地触摸文件一样(其名称就是这样得来的)。该程序的语法为：

```
touch [-acm] [-t time] file...
```

其中 *time* 是时间和日期，格式为[[YY]YY]MMDDhhmm[.ss]。

默认情况下，**touch** 同时将修改时间和访问时间设置为当前时间和日期。例如，假设文件 **essay** 上一次修改的时间是 7 月 8 日下午 2:30。您输入：

```
ls -l essay
```

输出为：

```
-rw----- 1 harley staff 4883 Jul 8 14:30 essay
```

现在是 12 月 21 日上午 10:30，您输入：

```
touch essay
```

接下来，当输入上述 **ls** 命令时将看到：

```
-rw----- 1 harley staff 4883 Dec 21 10:30 essay
```

什么时候使用 **touch** 命令呢？假设您准备分发一组文件——音乐、软件，以及其他任何文件，您都希望它们拥有相同的时间和日期。切换到存放这些文件的目录，然后输入：

```
touch *
```

所有匹配*通配符(参见第 24 章)的文件现在都拥有相同的修改时间和访问时间。

如果只希望改变修改时间，则可以使用 **-m** 选项。如果只希望改变访问时间，则可以使用 **-a** 选项。为了使用具体的时间和日期取代当前时间，可以使用 **-t** 选项，后面跟一个以格式[[YY]YY]MMDDhhmm[.ss]指定的时间。下面举两个例子。假设今天是 8 月 31 日，第一条命令(仅)将修改时间改变为当前日期的下午 5:29，第二条命令(仅)将访问时间改变为 2008 年 12 月 21 日上午 10:30：

```
touch -m -t 08311729 file1
touch -a -t 200812211030 file2
```

实际上,极少需要改变文件的修改时间或访问时间。但是, **touch** 拥有一个非常重要的副作用:如果指定的文件不存在,那么 **touch** 将创建这个文件。因此,每当需要时就可以使用 **touch** 命令创建全新的空文件。例如,为了创建文件 **newfile**,只需输入:

```
touch newfile
```

如果希望,则可以一次创建多个新文件:

```
touch data1 data2 data3 temp extra
```

当使用 **touch** 创建新文件时,修改时间和访问时间为当前时间和日期。如果这不符合您的要求,则可以在创建文件时使用上面讨论的选项设置一个具体的时间。

最后一个选项:如果您更新许多文件的修改时间或访问时间,而且不希望 **touch** 创建任何新文件,则可以使用 **-c**(no create, 不创建)选项。例如,下述命令将更新指定文件的时间。但是,如果文件不存在,那么该命令并不会创建该文件:

```
touch -c backup1 backup2 backup3 backup4
```

提示

大多数时候,不需要使用 **touch** 创建新文件,这是因为正如前面所述,新文件几乎总是在需要时自动被创建。

touch 真正发挥作用的地方就是当急需一些临时文件时(假设准备练习文件命令)。当发生这种情况时,使用 **touch** 就是创建一组全新空文件的最快方式,例如:

```
touch test1 test2 test3
```

25.2 命名文件

Unix 在文件的命名方面拥有极大的灵活性,它在这方面有两条基本的规则:

- (1) 文件名可以长达 255 个字符。^{*}
- (2) 文件名可以包含除/(斜线)及 null 字符之外的任何字符。

这样才合理。从第 24 章中可以知道, / 字符用作路径名中的分隔符,所以当然不能在文件名中使用它。null 字符是所有位都为 0 的字符(参见第 23 章)。该字符用作 C 编程语言中的字符串结束符号,最好永远不要在文件名中使用它。

除了这两条规则之外,我准备添加自己的第三条规则。

- (3) 创建对自己有意义的文件名。

^{*} 从技术上而言,文件名的最大长度由文件系统设置,而不是 Unix 或者 Linux。大多数现代文件系统的默认文件名最大长度为 255 个字符。但是,一些文件系统相当灵活。例如,如果您是一名文件系统极客,那么您可以轻易地修改 **ext2**、**ext3** 或者 **ext4** 文件系统,以允许文件名长达 1012 个字符。

例如，名称 **data** 相对于 **chemlab-experiment-2008-12-21** 来说就不是那么具有描述性。真实地讲，长名称比较复杂，而且键入比较麻烦，但是一旦知道了如何使用文件名自动补全(参见第 13 章)，那么您极少需要键入完整的名称。例如，对于上述长文件名，您可能只需键入 **ch<Tab>**，**shell** 就会自动地补全整个文件名。

我的建议就是在创建文件时，为所有的文件选取对自己有意义的文件名。否则，最终您会积累许多文件，这些文件可能包含也可能不包含有价值的信息。如果您像其他人一样，那么我可以确定，总有一天，您将不得不遍历所有的文件，以删除不需要的文件。当然，如果您和其他人一样，那么您或许永远不会真的这样做。*

提示

防止在目录中积累大量无用信息的最佳方式就是在创建文件时，为文件起一个有意义的名称。

Unix 允许创建包含所有奇特类型字符的文件名：退格、标点符号、控制字符，甚至是空格和制表符。很明显，这样的文件名将带来问题。例如，如果使用 **ls -l** 命令列举文件 **info;date** 的信息：

```
ls -l info;date
```

Unix 将把分号解释为两条命令的分隔符：

```
ls -l info
date
```

下面再举一个例子。假设您有一个文件，该文件的名称是 **-jokes**。那么在命令中使用该文件名有很大的麻烦，例如：

```
ls -jokes
```

Unix 将把 **-**(连字符)字符解释为选项的指示符。

一般而言，当文件名中包含拥有特殊含义的字符(<、>、|、!等)时，将会带来麻烦。最好的办法就是将文件名中使用的字符限制为不会引起误解的字符。图 25-1 中示范了可以在文件名中安全使用的字符。连字符可以使用，只要不把它们放在文件名的开头即可。

a、b、c……	小写字母
A、B、C……	大写字母
0、1、2……	数字
.	点
-	连字符
_	下划线

图 25-1 可以在文件名中安全使用的字符

Unix 允许在文件名中使用任何希望使用的字符，但是 **/**(斜线)或 **null** 除外。不过，如果坚持只使用字母、数字、点、连字符(不能位于文件名的开头)以及下划线的话，那么生活将减少很多麻烦。

* 如果您需要确认这一点，那么可以问问自己：“现在，我自己的计算机上有多少张照片？是不是有一天，我需要将它们分类呢？”

如果在文件名中包含有空格或者其他奇怪的字符，则可以通过引用名称使文件名可用(引用在第13章中解释过)。下面是一个特别华而不实的文件名的例子：

```
ls -l 'this is a bad filename, but it does work'
ls -l this\ is\ a\ bad\ filename\,\ but\ it\ does\ work
```

为了结束本节，我们准备介绍3条重要的文件命名约定。首先，正如第14章和第24章中讨论的，以.(点号)字符开头的文件称为点文件或者隐藏文件。当使用 **ls** 时，这样的文件只有在指定了 **-a**(all, 全部)选项时才能够列举出来。根据约定，我们使用以点号字符开头的名称作为包含配置数据或者初始化命令的文件的名称(图24-5中提供了一个常见的点文件列表)。

其次，我们经常使用以一个点号后面跟一个或多个字母结束的文件名来表示文件的类型。其中，C源文件的名称以 **.c** 结尾，例如 **myprog.c**；MP3音乐文件的名称以 **.mp3** 结尾；**gzip** 程序压缩的文件的名称以 **.gz** 结尾；等等。在这些情况中，这样的后缀称为**扩展名**。从字面上讲，扩展名有数百种不同类型。使用扩展名非常方便，因为它们允许使用通配符(参见第24章)引用一组文件。例如，可以使用下述命令列举目录中所有C源文件的名称：

```
ls *.c
```

最后，前面讲过，Unix区分大写字母和小写字母。因此，名称 **info**、**Info** 和 **INFO** 是完全不同的名称。Unix人士通常简单地使用 **info**。现在，考虑下述可能为包含程序或shell脚本的目录所使用的名称：

```
Program Files
ProgramFiles
programfiles
program-files
program_files
bin
```

第一个名称是 Windows 使用的名称。作为 Unix 用户，我们立即拒绝这个名称，因为它包含有空格。下一个名称中有两个大写字母，从而使这个名称难以键入，因此我们也拒绝这一名称。那么接下来的3个名称又如何呢？它们既不包含空格，也不包含大写字母和奇怪的字符。但是，对于重要的目录而言，使用简短的名称更加方便，因此我们建议使用 **bin** 作为该目录的名称。

在 Unix 世界中，我们有一个约定，即以大写字母开头的名称留给在某些方面特别重要的文件。例如，当下载以一组文件组织的软件时，通常会发现一个命名为 **README** 的文件。因为大写字母在 ASCII 码(参见第19章和第20章)中位于小写字母之前，所以这样的名称在目录列表中位于前面，也就是说可以突显出来^{*}。基于这一原因，建议在命名文件和目录时一般只使用小写字母。

^{*} 如果使用的是 C 区域设置，则是这种情况。如果使用的是 **en_US** 区域设置，那么情况就不是这样了(参见第19章)。

提示

如果您是程序员,那么您有时候很容易为正在开发的程序或 shell 脚本使用文件名 **test**。千万不要这样做。

shell 拥有一个名为 **test** 的内置命令,该命令用来比较 shell 脚本中的值。如果将自己的程序命名为 **test**,那么每当试图通过键入程序的名称运行程序时,运行的都将是 shell 的内置命令。运行过程中不会发生任何事情,这会导致您浪费大量的时间来查找问题。

(如果希望了解 **test** 程序的作用,可以查看联机手册。)

25.3 复制文件: cp

复制文件时使用的命令是 **cp**。该命令的语法为:

```
cp [-ip] file1 file2
```

其中 *file1* 是已有文件的名称, *file2* 是目标文件的名称。

该命令的使用相当简单。例如,如果您有一个文件 **data**,且希望为该文件复制一个副本 **extra**,则可以使用:

```
cp data extra
```

下面再举一个例子。您希望复制一份系统的口令文件(参见第 11 章)。副本文件称为 **pwd**,并且位于 **home** 目录中。正如第 24 章中所讨论的,~字符表示 **home** 目录,因此可以使用:

```
cp /etc/passwd ~/pwd
```

如果目标文件不存在,那么 **cp** 会创建该文件。如果目标文件已经存在,那么 **cp** 将替换这个文件。当发生这种情况时,没有办法再恢复被替换的数据。考虑第一个例子:

```
cp data extra
```

如果文件 **extra** 不存在,则创建这个文件。但是,如果文件 **extra** 已经存在,那么它将被替换。当发生这种情况时,原始文件中的数据将永远丢失,而且没有办法再找回这些数据(再阅读一遍最后一句)。

如果希望在文件的末尾追加数据,则不能使用 **cp** 程序。在这种情况下,可以使用 **cat** 程序,并重定向输出(参见第 16 章)。例如,下述命令将 **data** 的内容追加到 **extra** 的末尾。在这个例子中, **extra** 文件的原始内容被保留下来。

```
cat data >> extra
```

因为 **cp** 很容易清除文件的内容,所以如果希望额外小心的话,可以使用 **-i**(interactive, 交互)选项:

```
cp -i data extra
```

-i 选项告诉 **cp** 在替换已有文件之前进行询问。例如，有可能会看到如下提示：

```
cp: overwrite extra (yes/no)?
```

如果键入一个以字母 **y** 或 **Y**(代表“yes”)开头的答案，那么 **cp** 将替换文件。如果键入其他答案——例如按下<Return>键，那么 **cp** 将不替换文件。

我希望您知道的唯一一个另外的选项就是**-p**(**preserve**, 保持)。该选项使目标文件和源文件拥有相同的修改时间、访问时间以及权限(我们将在本章后面讨论权限)。

25.4 将文件复制到不同的目录中: **cp**

cp 命令可以用来将一个或多个文件复制到不同的目录中。其语法为：

```
cp [-ip] file... directory
```

其中 *file* 是已有文件的名称，*directory* 是已有目录的名称。**-i**(**interactive**, 交互)和**-p**(**preserve**, 保持)选项和上一节中描述的相同。

下面举例说明。为了将文件 **data** 复制到目录 **backups** 中，可以使用：

```
cp data backups
```

为了将 3 个文件 **data1**、**data2** 和 **data3** 复制到目录 **backups** 中，可以使用：

```
cp data1 data2 data3 backups
```

下面再举一个例子，一个稍微复杂一点的例子。假设您的工作目录是 **/home/harley/work/bin**。您希望将目录 **/home/harley/bin** 中的文件 **adventure** 复制到工作目录中。为了引用源目录，我们可以使用 **.././bin**；为了引用工作目录，可以使用 **.**。相应的命令为：

```
cp .././bin/adventure .
```

提示

在复制文件的过程中，可以使用通配符指定多个文件名(参见第 24 章)。例如，为了将文件 **data1**、**data2** 和 **data3** 复制到 **backups** 目录中，可以使用：

```
cp data[123] backups
```

如果没有其他以 **data** 开头的文件名，则可以使用：

```
cp data* backups
```

如果没有其他以 **d** 开头的文件名，则可以使用：

```
cp d* backups
```

25.5 将目录复制到另一个目录中: **cp -r**

通过使用 **-r** 选项, 可以让 **cp** 将目录及其所有文件复制到另一个目录中。其语法为:

```
cp -r [-ip] directory1... directory2
```

其中 *directory1* 是已有目录的名称, *directory2* 是目标目录的名称。-i(interactive, 交互)和-p(preserve, 保持)选项如本章前面所述。-r(recursive, 递归)选项告诉 **cp** 复制整个子树。

下面举例说明。假设在工作目录中, 您有两个子目录 **essays** 和 **backups**。在 **essays** 目录中, 还有许多文件和子目录。您输入:

```
cp -r essays backups
```

运行该命令之后, **essays** 的副本, 包括其中所有的文件和子目录, 现在都位于 **backups** 中。当使用 **-r** 选项时, **cp** 命令根据需要自动地创建新目录。

提示

为了复制目录中的所有文件, 可以用 * 通配符(参见第 24 章)使用 **cp**, 例如:

```
cp documents/* backups
```

为了复制目录本身以及其中包含的全部文件和子目录, 可以用 **-r** 选项使用 **cp**, 例如:

```
cp -r documents backups
```

25.6 移动文件: **mv**

要将文件移动到不同的目录中, 可以使用 **mv**(move, 移动)命令。其语法为:

```
mv [-if] file... directory
```

其中 *file* 是已有文件的名称, *directory* 是目标目录的名称。

mv 命令将一个或多个文件移动到已有目录中(如果要创建目录, 可以使用 **mkdir** 命令, 该命令在第 24 章中解释过)。下面举两个例子。第一个例子将文件 **data** 移动到目录 **archive** 中:

```
mv data archive
```

在移动文件时必须特别小心。如果目录 **archive** 不存在, 那么 **mv** 将认为您希望重命名文件(参见下面)。下面的例子将 3 个文件 **data1**、**data2** 和 **data3** 移动到目录 **archive** 中:

```
mv data1 data2 data3 archive
```

和大多数文件命令相同, **mv** 命令也可以使用通配符。例如, 上一条命令可以缩写为:

```
mv data[123] archive
```

如果文件已经在目标目录中存在，那么源文件将替换目标文件。在这些情况中，目标文件的原始内容将会丢失，而且没有办法恢复数据，因此一定要特别小心。如果希望小心从事以防数据丢失，可以使用**-i**(interactive, 交互)选项，例如：

```
mv -i data archive
```

这将告诉 **mv** 在替换已有文件之前进行询问。如果键入一个以字母 **y** 或 **Y**(代表“yes”)开头的答案，那么 **mv** 将替换文件。如果键入其他答案——例如按下<Return>键，那么 **mv** 将不替换文件。在这个例子中，**mv** 在替换文件 **archive/data** 之前将询问您是否许可。

与**-i**选项相反的选项是**-f**(force, 强制)。该选项将强制 **mv** 在不询问的情况下替换文件。**-f**选项将忽略**-i**选项以及文件权限(本章后面解释)强加的限制。使用**-f**时要特别小心，只有在非常清楚自己在做什么的时候才能使用。

25.7 重命名文件或目录：mv

重命名文件或目录时，可以使用 **mv**(move, 移动)命令。其语法为：

```
mv [-if] oldname newname
```

其中 *oldname* 是已有文件或目录的名称，*newname* 是新名称。**-i**(interactive, 交互)和**-f**(force, 强制)选项与上一节中描述的相同。

重命名文件或目录相当简单。例如，为了将文件 **unimportant** 重命名为 **important**，可以使用：

```
mv unimportant important
```

如果目标文件(在这个例子中为 **important**)已经存在，那么目标文件将被替换。目标文件中的所有原始数据都将丢失，而且没有办法恢复，因此输入命令时一定要小心。**mv** 命令还可以使用上一节中描述的**-i**和**-f**选项来控制文件的替换：**-i**告诉 **mv** 在替换文件之前询问；**-f**不加考虑强制进行替换。

使用 **mv** 可以同时重命名和移动文件。例如，假设 **incomplete** 是一个文件，**archive** 是一个目录。下述命令将 **incomplete** 移动到目录 **archive**(这个目录必须存在)。作为移动过程的一部分，文件将被重命名为 **complete**：

```
mv incomplete archive/complete
```

最后，考虑对目录 **old** 使用如下 **mv** 命令会发生什么事情：

```
mv old new
```

如果目录 **new** 不存在，那么目录 **old** 将被重命名为 **new**。但是，如果目录 **new** 已经存在，那么目录 **old** 将移入并成为目录 **new** 的一个子目录(好好地想一想这一点)。

25.8 删除文件：rm

为了删除文件，需要使用 **rm**(remove, 移除)命令。该命令的语法为：

```
rm [-fir] file...
```

其中 *file* 是希望删除的文件的名称。

(注意该命令的名称是“remove, 移除”，而不是“delete, 删除”。当在本章后面讨论链接时就会明白其中的含义。)

删除文件时，只需指定文件的名称。下面举一些例子。第一条命令删除工作目录中的文件 **data**。第二条命令删除工作目录中的文件 **essay**。最后一条命令删除目录 **bin** 中的文件 **spacewar**，而 **bin** 目录位于工作目录中。

```
rm data
rm ~/essay
rm bin/spacewar
```

和所有的文件命令一样，**rm** 命令也可以使用通配符(参见第 24 章)。下面举两个例子。第一条命令删除工作目录中的文件 **data1**、**data2** 和 **data3**。第二条命令删除工作目录中除点文件之外的所有文件(很明显，这是一条功能非常强大的命令，因此不要练习使用这条命令)。

```
rm data[123]
rm *
```

一旦文件被删除，它就永远消失了。没有任何办法可以找回删除的文件，因此一定要小心。

当在 **rm** 命令中使用通配符时，最好先使用 **ls** 命令测试一下，查看匹配哪些文件。下面举一个例子。假设您希望删除文件 **data.backup**、**data.old** 和 **data.extra**。您准备使用通配符表达式 **data*** 来删除这些文件，而 **data*** 匹配所有以 **data** 开头的文件。但是，为了小心起见，您输入了下述命令来检查这一表达式：

```
ls data*
```

输出为：

```
data.backup data.extra data.important data.old
```

您发现自己忘记了文件 **data.important**。如果直接对 **rm** 命令使用 **data*** 通配符表达式的话，您将失去这个文件。作为替代，您可以使用：

```
rm data.[beo]*
```

这将只匹配那些希望删除的文件。

提示

在 **rm** 命令中使用通配符时，一定要先使用 **ls** 命令确认匹配哪些文件。

25.9 如何防止误删文件: `rm -if`

正如上一节中所指出的, 在使用 `rm` 命令删除文件之前, 最好先使用 `ls` 命令检查一下通配符模式是否正常。但是, 即便使用 `ls` 检查了通配符模式, 在输入 `rm` 命令时, 仍然有可能发生键入错误。因此下面给出一种解决这一问题的方法, 该方法十分安全。

在第 13 章讨论别名时, 我们示范了如何定义别名 `del`, 该别名使用和其前面的 `ls` 命令相同的参数运行 `rm` 命令。对于 Bourne shell 家族(Bash、Korn shell)来说, 定义该别名的命令为:

```
alias del='fc -s ls=rm'
```

对于 C-Shell 家族(Tcsh、C-Shell)来说, 定义该别名的命令为:

```
alias del 'rm \!ls:*
```

(详细细节在第 13 章中解释过。)为了永久地定义别名 `del`, 只需在环境文件(参见第 14 章)中放入合适的命令。一旦定义了别名, 别名的使用将非常方便。首先, 输入 `ls` 命令, 并对 `ls` 使用描述希望被删除的文件的通配符表达式。例如:

```
ls data.[beo]*
```

检查文件列表。如果所列出的文件确实都是希望删除的文件, 则可以输入:

```
del
```

这将使用上述 `ls` 命令列出的文件名执行 `rm` 命令。如果文件列表并不是自己所希望的, 则可以尝试修改模式。

另一种简单的做法就是使用 `-i`(interactive, 交互)选项。该选项告诉 `rm` 在删除每个文件之前先请求许可。例如, 您可以输入:

```
rm -i data*
```

`rm` 程序将为每个文件显示一条消息, 请求继续执行, 例如:

```
rm: remove regular file `data.backup'?
```

如果您键入一个以“y”或“Y”(代表“yes”)开头的响应, 那么 `rm` 将删除文件。如果您键入了其他任何答案——例如<Return>键, 那么 `rm` 将保留文件。

常见的做法是定义一个别名, 在每次使用 `rm` 命令时都自动地插入 `-i` 选项。下面就是相应的别名。第一个别名适用于 Bourne Shell 家族, 第二个别名适用于 C-Shell 家族:

```
alias rm='rm -i'
alias rm 'rm -i'
```

一些系统管理员将这样的别名放在系统级的环境文件中, 认为这样可以帮用户的忙。这种做法是不可取的, 之所以这样说, 有两个原因。第一, Unix 的特点是简洁而准确。

每次删除文件都要键入一个 **y** 会减缓思考速度。如果使用自动的 **-i** 选项，就会使人有依赖该选项的想法，以至于变得粗心大意。

如果您不同意这种看法，则可以这样想：的确，对于一个还没有习惯使用 **rm** 命令的新手来说，第一周可能偶然误删一两个文件，而且还没有办法恢复文件。但是，经验是一个重要的因素，因此不用多久，新用户就会知道要小心地使用该命令。我深信，从长远来看，提供应用技术总比助长人的依赖性要好得多。事实的真相是，不管 **rm** 的潜在功能是好是坏，有经验的 Unix 用户极少误删文件，因为他们形成了良好的习惯。

之所以不自动设置 **-i** 选项的第二条原因是，终究不是每个人仅使用一种 Unix 或 Linux 系统。当人们习惯了慢腾腾的、笨拙的并且删除每个文件都要请求许可的 **rm** 命令之后，就会忘记并非所有的 Unix 系统都是这样。有一天，当使用不同的系统时，他们很容易就会犯这种灾难性的错误。手指拥有记忆功能，一旦习惯了键入 **rm**，而不是 **rm -i**，那么忘记这种习惯就比较难。

基于这一原因，如果必须为 **rm -i** 创建别名，最好定义一个不同的名称，例如：

```
alias erase='rm -i'
alias erase 'rm -i'
```

下面介绍最后一点。在本章后面，我们将讨论文件的权限。那时，您将会知道有 3 种类型的权限：读、写和执行。现在不做详细讨论，只是说明一点：如果没有写权限，就不能删除文件。如果试图删除一个没有写权限的文件，那么 **rm** 将请求许可，在用户许可的情况下系统将忽略文件权限保护机制。

例如，假设文件 **data.important** 的权限是 **400**（“400”的含义将在后面解释。实际上，它意味着您只拥有这个文件的读权限，没有写权限和执行权限）。输入命令：

```
rm data.important
```

您将看到下述问题：

```
rm: remove write-protected regular file `data.important'?
```

如果键入以“y”或“Y”（代表“yes”）开头的响应，那么 **rm** 将删除这个文件。如果键入其他任何答案——例如<Return>键，那么 **rm** 将保留该文件。如果您特别仔细，那么您可以使用 **-f**(force, 强制)选项告诉 **rm** 执行删除，不必请求许可——即不考虑文件权限：

```
rm -f data.important
```

在一些系统上，**-f** 选项将覆盖 **-i** 选项。

提示

当使用 **rm** 删除文件时，**-f**(force, 强制)选项将忽略文件权限和 **-i** 选项(某些系统上)。基于这一原因，只有当确信自己在做什么事情时，才可以使用 **-f** 选项。

25.10 删除整个目录树: `rm -r`

删除整个目录树, 需要使用带 `-r`(recursive, 递归)选项的 `rm` 命令, 并指定目录名。这将告诉 `rm` 不仅要删除目录, 而且还要删除这个目录中的所有文件和子目录。例如, 假设您有一个目录 `extra`。在这个目录中, 有许多文件和子目录。在每个子目录中还有许多文件和子目录。为了一次将这些内容全部删除, 可以使用:

```
rm -r extra
```

下面再举一个例子, 一个不太切合实际、但很强大的简单例子。为了删除工作目录中的所有内容, 可以使用:

```
rm -r *
```

很明显, `rm -r` 是一条危险的命令, 因此如果对自己的操作有丝毫不能确信的话, 最好不要使用 `-r` 选项。最低限度, 应该同时考虑使用 `-i`(interactive, 交互)选项。这将告诉 `rm` 在删除每个文件和目录之前请求许可, 例如:

```
rm -ir extra
```

如果希望快速而又平静地删除整个目录树, 则可以加上 `-f` 选项:

```
rm -fr extra
```

记住, 在一些系统上, `-f` 选项将覆盖 `-i` 选项, 因此在使用 `-f` 选项时一定要小心。

提示

在使用带 `-r` 选项的 `rm` 命令删除整个目录树之前, 一定要好好地想一想, 并使用 `pwd` 显示工作目录。如果您位于错误的目录中, 想一想下述命令会产生什么样的后果:

```
rm -rf *
```

最后, 为了对 `rm` 命令做一总结, 我们再看看该命令如何轻易地清除所有的文件。假设您的 `home` 目录中包含许多子目录, 这是您数月以来辛苦劳动的成果。您希望删除 `extra` 目录下的所有文件和子目录。

在进行删除时, 您并没有位于 `home` 目录中。您需要做的就是先切换到 `home` 目录中, 然后输入 `rm` 命令:

```
cd
rm -fr extra
```

但是, 您自己这样想: “没有必要键入两条命令, 我可以用一条命令完成全部事情。” 您希望输入:

```
rm -fr ~/extra
```

(记住, 正如第 24 章中讨论的, `~` 字符表示 `home` 目录。)但是, 匆忙中您一不小心在斜

线前面键入了一个空格：

```
rm -fr ~ /extra
```

结果，您输入了一条删除两个目录树的命令，~(home 目录)和/extra 目录中的所有文件都将被删除。

一旦按下了<Return>键，那么试图通过按^C 或<Delete>(无论使用哪一个都代表 **intr** 键)来终止这个命令就不管用了。因为计算机要比您快许多，没有什么办法能够停止已经运行的 **rm** 命令了。当您意识到发生什么情况时，所有的文件都已经不见了，包括点文件(我测试过这条命令，因此希望您不要这样做：相信我)。

正如第 4 章中讨论的，当以 **root** 登录时，就会成为超级用户。作为超级用户，能够执行任何操作，包括删除整个系统中的任何文件或目录。如果以超级用户登录并输入下述命令，您认为会发生什么事情呢？

```
rm -fr /
```

(除非得到父母的许可，否则不要在家里这样做。)

技术提示

当使用 **rm -fr** 处理变量时一定要特别小心。例如，假设您有一个 shell 脚本，该脚本按如下方式使用变量 **\$HOME** 和 **\$FILE**：

```
rm -fr $HOME/$FILE
```

如果由于某些原因，这两个变量都没有定义，那么该命令将变成：

```
rm -fr /
```

运气好的话，您将只删除自己的所有文件，包括所有的(隐藏)点文件。但是，如果以超级用户运行该脚本的话，那么后果将是灾难性的。添加 **-i** 选项也没有什么帮助，因为正如前面所解释的，在许多系统上，**-f** 选项将覆盖 **-i** 选项。

25.11 被删除文件恢复的可能性

没有可能。

25.12 文件权限

Unix 为每个文件维护一组文件权限(file permission)，通常称为权限(permission)。这些权限控制哪些用户标识可以访问文件，以及以何种方式访问文件。权限有 3 种，包括读权限(read permission)、写权限(write permission)和执行权限(execute permission)。这 3 种权限

相互之间彼此独立。例如，您的用户标识可能只拥有某个特定文件的读和写权限，但没有该文件的执行权限。这里重要的是要理解，权限是与用户标识关联的，而不是与用户。例如，如果有人以您的用户标识登录，那么在他访问您的文件时，他拥有和您相同的权限。

文件权限的准确含义依赖于文件的类型。对于普通文件来说，权限的含义相当直接：读权限允许用户标识读取该文件，写权限允许用户标识写该文件，执行权限允许用户标识执行该文件。当然，对于一个不可执行的文件来说，拥有执行权限没有任何意义。通常，如果文件是程序或者某种类型的脚本，那么它就是可执行的。例如，`shell` 脚本中就包含有可由 `shell` 执行的命令。

3 种类型的权限是相互区别的，但是可以结合使用。例如，为了修改一个文件，就需要同时具有该文件的读权限和写权限。为了运行 `shell` 脚本，就需要同时拥有读权限和执行权限。

本章后面将会介绍，您能够为自己的文件设置和修改权限。这样做有两点原因：

- 限制其他用户的访问

通过限制哪些用户标识可以访问自己的文件，以一种相当直接的方式为数据提供安全。

- 避免自己错误使用

如果希望保护文件，阻止其被不小心误删，则可以不设置文件的写权限。许多替换或删除数据的命令在改变没有写权限的文件之前都会请求确认(本章前面讨论的 `rm` 和 `mv` 命令就是这种情况)。

对于目录而言，目录的权限和普通文件有所不同。读权限允许用户标识读取目录中的文件名。写权限允许用户标识修改目录(创建、移动、复制、删除)。执行权限允许用户搜索目录。

如果只拥有读权限，则只能列举目录中的文件名，仅此而已。除非拥有执行权限，否则不能查看文件的大小、查看子目录或者使用 `cd` 改变目录。

下面考虑一个不寻常的组合。如果您拥有一个目录的写权限和执行权限，但是没有该目录的读权限，那会怎么样呢？这样的话，您能够访问并修改该目录，但不能读取它。因此，您不能列举该目录的内容，但是，如果知道文件的名称，则可以删除该文件。

出于参考目的，图 25-2 列举了适用于普通文件和目录的各种文件权限。

普通文件	
读	读取文件
写	写入文件
执行	执行文件
目录	
读	读取目录
写	创建、移动、复制或删除目录条目
执行	搜索目录

图 25-2 文件权限一览表

文件权限控制用户标识对文件的访问。每个文件有 3 组权限：一组针对属主、一组针对组、一组针对其他用户。每组权限有 3 部分：读权限、写权限和执行权限。这些权限的含义对普通文件和目录来说有所不同。

提示

刚开始接触目录权限时，可能不太好理解它。在本章后面的学习中，就会知道一个目录条目仅包含一个文件名，以及一个指向该文件的指针，并不包含实际的文件。

一旦理解了这一点，就能够很好地理解目录的权限。读权限意味着可以读取目录条目。写权限意味着可以改变目录条目。执行权限意味着可以使用目录条目。

25.13 setuid

在 Unix 系统中，您作为个人，根本不存在。您以特定的用户标识登录，然后运行与该用户标识绑定的程序。作为生活在外部世界中的人来说，您的角色就限制在输入的提供以及输出的阅读上。真正完成工作的不是您，而是您的程序。例如，为了排序数据，您使用 **sort** 程序；为了重命名文件，使用 **mv** 程序；为了显示文件，使用 **less** 程序；等等。

通常(只有一个例外，我们稍后再进行讨论)，每当运行程序时，该程序将在您的用户标识的授权下运行。这意味着您的程序拥有和用户标识相同的权利。例如，假设您的用户标识没有文件 **secrets** 的读权限。您希望查看这个文件的内容，因此您输入了：

```
less secrets
```

因为您的用户标识不能读取该文件，所以您所调用的来完成这一任务的程序也不能读取该文件。结果是您将看到如下消息：

```
secrets: Permission denied
```

如果确实希望查看 **secrets** 文件的内容，那么有 3 种选择。第一，您可以改变 **secrets** 文件的文件权限(本章后面解释)。第二，您可以用一个拥有 **secrets** 文件的读权限的用户标识登录。第三，如果您知道 **root** 账户的口令，则可以以超级用户登录，从而绕过几乎所有的限制。

换句话说，除非您是超级用户，否则您的程序将受用户标识的限制和约束。但是有一个例外，有时候，普通的用户标识有可能需要以特殊的权限运行程序。为了使这一点成为可能，人们设计了一个特殊的文件权限设置，以允许其他用户标识访问文件，就好像他们是文件的属主(创建者)一样。这个特殊的权限称为 **setuid**(发音为“set U-I-D”)或者 **suid**。该名称代表“set userid，设置用户标识”。

在大多数情况中，**setuid** 用来允许普通用户标识运行从 **root** 拥有的程序中挑选的程序。这意味着无论哪个用户标识运行程序，它都以 **root** 的特权运行。这样就允许程序完成通常由超级用户执行的任务。例如，为了修改口令，需要使用 **passwd** 程序。但是，要修改口令，该程序必须修改口令文件和影子文件(参见第 11 章)，而这两个文件需要超级用户的特权。基于这一原因，**passwd** 程序本身存储在一个由 **root** 拥有的文件中，并且打开了 **setuid**。

那么如何断定文件拥有 **setuid** 文件权限呢？当显示文件的长列表时，将会看到这些文件的文件权限中的字母“x”被字母“s”所取代。例如，您输入：


```
ls -l /usr/bin/passwd
```

这将显示 `passwd` 程序的一个长列表，输出为：

```
-r-s--x--x 1 root root 21944 Feb 12 2007 /usr/bin/passwd
```

文件权限中的“s”(从左边数第4个字符)表示 `setuid` 权限。“s”取代了“x”。

很明显，这样的权限可能存在安全风险。毕竟，需要以超级用户特权运行的程序可以用来攻入系统或者进行破坏。因此，`setuid` 的使用是受严格限制的。通常，`setuid` 只用来允许普通用户标识为了执行某个具体的任务而以临时特权来运行程序。

25.14 Unix 维护文件权限的方式：id、groups

现在使用的文件权限的组织方式仍然是 Unix 系统的最初开发者——贝尔实验室的程序员们设计的。在 Unix 开发时，贝尔实验室的人们都是以小组为单位进行工作的，他们共享程序和文档。基于这一原因，Unix 开发人员定义了3种类别的人：用户、用户组以及系统中的每个人。然后他们设计 Unix，为每个文件维护3组权限。下面进行详细介绍。

创建文件的用户标识就是文件的属主。属主是可以改变文件权限的唯一用户标识*。第一组文件权限描述属主如何访问文件。每个用户都属于一个组(下面解释)，第二组权限应用于和属主位于同一组中的所有其他用户标识。第三组权限应用于系统上的其他用户标识。这意味着，对于每个文件和目录，可以为自己、组中的其他人以及剩下的每个人单独分配读、写和执行权限。

下面举例说明。您和一组人正在开发程序。包含程序的文件位于您个人的某个目录中。您可以设置文件权限，使您和您的小组都拥有读、写以及执行权限，而系统上的其他用户只有读和执行权限。这意味着，尽管每个人都可以运行程序，但是只有您或您所在的小组中的成员可以修改程序。

下面再举一个例子。您拥有一个文档，且不希望其他人看到。在这种情况下，只需授予您自己读和写权限，而不把读和写权限授予您所在组的成员和系统上的其他人。**

这里，重要的是要理解适用于“每个人”的权限并不包括您自己和您自己所在组中的成员。想象一个奇怪的场景，在这个场景中，您将文件的读权限授予每个人，但是没有为组中成员授予任何权限。这时，组中的成员无法读文件，但是，其他人都可以。另外，如果网络中的用户访问您的文件系统，那么他们也属于“每个人”的类别范畴，即使他们在该系统中没有账户。

那么，哪些人应该位于组中呢？当系统管理员为您创建账户时，他也为您分配了一个组。就像每个用户有一个用户标识一样，每个组也拥有一个组标识。系统中组标识的列表存放在文件 `/etc/group` 中，该文件可以自由查看：

* 唯一的例外就是超级用户，超级用户可以完成几乎所有的东西，可以改变任何文件的权限。如果需要，超级用户还可以通过分别使用 `chown` 和 `chgrp` 命令改变文件的属主和组。

** 但是，请记住，您无法对超级用户隐藏任何东西。

```
less /etc/group
```

组的名称存放在口令文件/etc/passwd(第 11 章中描述)中, 和用户标识、home 目录的名称以及其他信息存放在一起。显示用户标识和组标识的最简单方法就是使用 **id** 命令(只需键入命令本身即可, 不需要任何选项)。

id 命令在一个非常特殊的环境中非常方便。这个环境就是, 在进行系统管理工作时, 不时地需要从自己的用户标识切换到 **root**(超级用户), 过一会再切回。如果您被搞迷惑了, 不知道使用的是哪个用户标识, 就可以输入 **id** 命令(这种情况我每个月都要遇到几次)。

问题: 假设系统管理员走过计算机机房, 看到一台用户已经离开, 且登录到 Unix 系统的机器。他该怎么办呢?

答案: 他所做的第一件事就是输入 **id** 命令, 查看是谁登录的。然后为该用户标识留一个便条——告诉该用户要注意一点, 最后键入 **exit** 注销系统。

在 Unix 的早期, 每个用户标识仅属于一个组。但是, 现代的 Unix 系统允许用户同时属于多个组。对于每个用户标识来说, 在口令文件中列出来的组称为主组(primary group)。如果用户标识还属于其他组, 那么这些组称为辅组(supplementary group)。显示用户所属组全部列表的方式有两种。第一种就是使用 **id** 程序(在 Solaris 系统中必须使用 **ls -a**):

```
id
```

下面是一些示例输出:

```
uid=500(harley) gid=500(staff) groups=500(staff),502(admin)
```

在这个例子中, 用户标识 **harley** 属于两个组: 主组 **staff** 和一个辅组 **admin**。

另一种显示所有组的方式是使用 **groups** 程序。该程序的语法为:

```
groups [userid...]
```

其中 **userid** 是用户标识。

默认情况下, **groups** 显示当前用户标识所属的组的名称。如果指定一个或多个用户标识的话, 那么 **groups** 将显示这些用户标识所属的组。在自己的系统中试试下述两条命令。第一条命令显示您自己所属的全部组; 第二条命令显示超级用户(用户标识 **root**)所属的组:

```
groups
groups root
```

组有多重要呢? 在 20 世纪 70 年代, 组非常重要。大多数 Unix 用户都是工作在值得信任的环境中的研究人员, 这类环境不与外部网络连接。将每个用户标识放在一个组中, 可以让研究人员共享工作, 并与同事们协同工作。但是, 现在, 对于普通用户来说, 基于下述两个原因, 组有时候被忽略了。

首先, 大多数人拥有自己的 Unix 或 Linux 计算机, 而且当您是所使用系统的唯一用户时, 没有其他人可以共享。其次, 即使在共享系统或大型网络上, 系统管理员也会发现维护一个足够小且有用的组并不值得。例如, 如果您是大学的研究生, 那么您的用户标识可能处于某个大组(比如说社会科学系的全部学生组成的组)内, 与这些人共享没有实际意义。

但是，您也应该知道一些组织确实不辞辛苦地维护组，以便共享数据文件或可执行程序。例如，在大学里，可以向学习特定课程的学生授予为这门课程所建立的组中的用户标识。通过这种方式，教师可以创建只能由这些学生访问的文件。

提示

除非确实需要与组中的其他用户共享文件，否则最好完全忽略组这一思想。当设置文件权限(本章后面解释)时，只需将“组”设置成和“每个人”拥有相同的权限即可。

25.15 显示文件权限：ls -l

显示文件的文件权限时，需要使用带-l(long listing, 长列表)选项的ls命令。权限在输出的左边显示。显示目录的权限时，在设置-l选项的同时还要设置-d选项。(ls命令以及这些选项的解释参见第24章)。

下面举例说明。您输入下述命令，查看工作目录中的文件：

```
ls -l
输出为：
total 109
-rwxrwxrwx 1 harley staff 28672 Sep 5 16:37 program.allusers
-rwxrwx--- 1 harley staff 6864 Sep 5 16:38 program.group
-rwx----- 1 harley staff 4576 Sep 5 16:32 program.owner
-rw-rw-rw- 1 harley staff 7376 Sep 5 16:34 text.allusers
-rw-rw---- 1 harley staff 5532 Sep 5 16:34 text.group
-rw----- 1 harley staff 6454 Sep 5 16:34 text.owner
```

我们已经在第24章中讨论了这种输出的大部分内容。下面简要概括一下，文件名位于最右边。向左边看，依次显示的是上次修改的时间和日期、文件大小(以字节为单位)以及属主的组和用户标识。在这个例子中，文件的属主是用户标识harley，组为staff。文件属主的左边是链接(在本章后面进行讨论)的数量。在最左边，每行的第一个字符是文件类型指示符。普通文件标识为-(连字符)，目录(这个例子中没有)标识为d。

下面我们希望关注的内容是文件类型指示符右边的9个字符。它们的含义如下所示：

r = 读权限
w = 写权限
x = 执行权限
- = 没有权限

在分析文件的权限时，将9个字符简单地分成3组。从左向右，这些组分别显示文件属主、组以及系统上所有其他用户标识的权限。下面分析示例中所有文件的权限：

Owner	Group	Other	File
rwx	rwx	rwx	program.allusers
rwx	rwx	---	program.group

```

rwx  ---  ---  program.owner
rw-  rw-  rw-  text.allusers
rw-  rw-  ---  text.group
rw-  ---  ---  text.owner
    
```

现在我们可以准确地知道每个权限是如何分配的了。例如，对于文件 `text.owner` 来说，文件属主拥有读和写权限，组和其他用户标识没有任何权限。

25.16 文件模式

Unix 使用一个紧凑的 3 位数字的代码来表示一个完整的文件权限集。该代码称为文件模式(file mode)，或简称为模式(mode)。例如，上例中文件 `text.owner` 的模式为 `600`。

在模式中，每位数字代表一个权限集。第一位数字表示拥有该文件的用户标识的权限，第二位数字表示组中各用户标识的权限，第三位数字表示系统中其他所有用户标识的权限。对于前面刚提及的例子来说，情况如下所示：

- 6 = 属主的权限
- 0 = 组的权限
- 0 = 其他所有用户标识的权限

下面说明该代码形成的方式。首先介绍代表各种权限的数字值，如下所示：

- 4 = 读权限
- 2 = 写权限
- 1 = 执行权限
- 0 = 没有权限

对于每一组权限来说，只需将适当的数字加在一起即可。例如，为了表示读和写权限，只需将 `4` 和 `2` 加在一起。图 25-3 列出了每种可能的组合以及相应的数字值。

读	写	执行	分量	和
—	—	—	0+0+0	0
—	—	有	0+0+1	1
—	有	—	0+2+0	2
—	有	有	0+2+1	3
有	—	—	4+0+0	4
有	—	有	4+0+1	5
有	有	—	4+2+0	6
有	有	有	4+2+1	7

图 25-3 文件权限组合及相应的数字值

文件权限有 3 种类型：读权限、写权限和执行权限。这些权限的值由 3 个不同的数字表示，这 3 个数字加在一起形成表示权限的值，如表中所示。详情请参见正文。

下面举例说明。假设要计算满足下面条件的文件的模式：

- 属主拥有读、写和执行权限；
- 组拥有读和写权限；
- 其他所有用户只拥有读权限。

属主：读+写+执行 = $4+2+1 = 7$

组： 读+写 = $4+2+0 = 6$

其他： 读 = $4+0+0 = 4$

因此，该文件的模式为 **764**。下面再看一看上一节中的例子：

Owner	Group	Other	Mode	File
rw x = 7	rw x = 7	rw x = 7	777	program.allusers
rw x = 7	rw x = 7	--- = 0	770	program.group
rw x = 7	--- = 0	--- = 0	700	program.owner
rw - = 6	rw - = 6	rw - = 6	666	text.allusers
rw - = 6	rw - = 6	--- = 0	660	text.group
rw - = 6	--- = 0	--- = 0	600	text.owner

现在，我们再举一个相反的例子。文件模式为 **540** 的文件权限如何呢？根据图 25-3 所示，我们知道：

属主： **5** = 读+执行

组： **4** = 读

其他： **0** = 没有权限

因此，文件的属主可以读和执行文件，组用户只能读文件，其他人没有该文件的权限。

25.17 改变文件权限

改变文件的权限时，需要使用 **chmod**(change file mode, 改变文件模式)命令。该命令的语法为：

```
chmod mode file...
```

其中 *mode* 是新文件模式，*file* 是文件或目录的名称。

只有属主和超级用户才可以改变文件的文件模式。正如前面所述，Unix 自动使您成为您自己所创建的每个文件的属主。

下面举一些使用 **chmod** 命令的例子。第一条命令改变指定文件的模式，给予文件属主读和写权限，给予组和其他任何人的只有读权限。这些权限适合于希望他人阅读，但不能进行修改的文件。

```
chmod 644 essay1 essay2 document
```

接下来的一条命令给予文件属主读、写和执行权限，给予组和其他任何人读和执行权限。这些权限适合于包含有希望他人执行，但不能进行修改的程序的文件：

```
chmod 755 spacewar
```

通常，除非有特殊理由，否则最好限定其他人的权限。下述两条命令示范如何只对文件属主设置权限，而组和其他任何人没有权限。第一条命令只设置读和写权限：

```
chmod 600 homework.text
```

第二条命令设置读、写和执行权限：

```
chmod 700 homework.program
```

当创建 shell 脚本或程序时，默认情况下，只拥有脚本和程序的读和写权限。为了执行脚本，必须添加执行权限。使用的命令为 **chmod 700**(如果希望共享的话，则可以使用 **chmod 755**)。

提示

为了避免问题，不要给不可执行的文件授予执行权限。

25.18 Unix 为新文件指定权限的方式：umask

当 Unix 创建新文件时，将根据文件的类型为文件指定下述几种模式：

666：不可执行的普通文件

777：可执行的普通文件

777：目录

在这一初始模式上，Unix 再减去用户掩码(user mask)值。用户掩码是一种模式，由自己设置，表明希望限制的权限。设置用户掩码时，需要使用 **umask** 命令。该命令的语法为：

```
umask [mode]
```

其中 *mode* 指定希望限制的权限。

最好将 **umask** 命令放在登录文件中，以便每次登录时都可以自动地设置用户掩码。实际上，第 14 章中示范的示例登录文件中就有一条 **umask** 命令。

那么用户掩码应该如何设置呢？下面考虑一些例子。首先，假设您希望抑制组和其他任何人的写权限。这种情况下，可以使用模式 **022**：

```
umask 022
```

该用户掩码将共享文件，但是不让其他人改变文件。大多数情况下，最谨慎的策略就是尽可能使文件是专用的。这种情况下，可以抑制组和其他所有人的所有权限——读、写和执行。这时，使用的模式为 **077**：


```
umask 077
```

为了显示用户掩码的当前值，可以输入没有参数的 **umask** 命令：

```
umask
```

注意：**umask** 命令本身是一条内置命令，这意味着该命令的准确行为依赖于所使用的 shell。一些 shell 不显示前导 0。例如，如果用户掩码是 **022**，则看到的可能是 **22**；如果用户掩码是 **002**，则看到的可能是 **2**。如果自己的 shell 是这种情况，就假设有 0 在前面。

提示

除非有很好的理由，否则应该将 **umask 077** 命令放入登录文件，从而使文件完全专有。如果希望共享文件，则可以对文件使用 **chmod** 命令。

25.19 清空文件内容：shred

正如本章前面所讨论的，一旦删除文件，就没有办法再找回这个文件。但是，文件所使用的实际磁盘空间还没有被清除。文件系统只是将这部分磁盘空间标识为可以重用。最终，这部分磁盘空间将被重用，旧数据将被新数据覆盖。在忙碌的大型 Unix 系统中，这可能只需几秒钟时间。但是，无法确定这种情况何时会发生，有时候旧数据可能会在磁盘的未使用部分隐藏很长一段时间。实际上，有一些特殊的“恢复删除”工具能够查看磁盘未使用的部分，并恢复旧数据。

此外，即使数据被覆盖了，在极端的情况下数据也有可能恢复，只要数据没有被多次覆盖。如果将硬盘拿到拥有非常昂贵的数据恢复设备的实验室中去，则有可能通过分析硬盘磁面的磁迹恢复硬盘上被覆盖过的旧数据。

对于真正的偏执狂来说，永远删除文件的最佳方式就是毁坏存储介质。对于 CD 或软盘来说，这种方法相对容易，但是对于硬盘来说，就没有那么容易了，特别是当希望清除少数几个文件，而不是整个硬盘时。因此，对于那些极其罕见的情况，即简单的文件删除不能满足要求，GNU 实用工具(参见第 2 章)提供了一个程序来完成这种任务，该程序就是 **shred**。尽管 **shred** 程序并不是普遍存在的，但是大多数 Linux 系统都提供该程序。该程序的语法为：

```
shred -fvuz [file...]
```

其中 *file* 是文件的名称。

shred 程序的目的是多次覆盖硬盘上已有的数据，从而使世界上最昂贵的数据恢复设备也难以记录硬盘磁面的磁迹。我们所需做的就是指定一个或多个文件的名称，然后 **shred** 就会自动完成工作。如果在使用 **shred** 程序时添加 **-v(verbose, 详细)** 选项，那么 **shred** 在处理过程中将显示处理消息：

```
shred -v datafile
```

默认情况下, **shred** 将覆盖数据多次, 并且使该文件包含的是随机的数据。当然, 随机数据只是说明该文件已经被“粉碎”。为了隐藏这一点, 可以使用 **-z** 选项, 以告诉 **shred** 在结束任务时将文件全部填充为 0。进一步讲, 如果希望在处理之后删除文件, 则可以使用 **-u** 选项。最后, 为了忽略受限制的文件权限, 可以使用 **-f(force, 强制)** 选项。

下面是终极的 **shred** 命令。该命令将忽略已有的文件权限, 通过覆盖文件多次清除全部数据, 再将文件全部填充成 0, 然后再删除文件:

```
shred -fuvz datafile
```

当然, **shred** 程序只能做到这一点。如果文件已经自动备份到另一个系统上, 或者复制到镜像站点上, 那么世界上所有的粉碎工具都无法清除远端的副本。此外, **shred** 并不是适用于所有的文件系统。例如, 当更新 ZFS 文件系统(由 Sun 公司开发)上的文件时, 新数据将写到磁盘上的另一个位置上。直到旧数据所在的那一部分硬盘空间被另一个文件所重用, 旧数据才被替换。

25.20 链接的概念: stat、ls -i

当 Unix 创建文件时, Unix 完成两件事情。第一, Unix 在存储设备上保留一块空间用来存储数据。第二, Unix 创建一个称为索引节点(index node)或 i 节点(“i-node”)的结构, 来存放文件的基本信息。i 节点包含使用文件所需的全部文件系统信息。图 25-4 列举了典型 Unix 文件系统中 i 节点所包含的内容。普通用户不必了解 i 节点的内容, 因为文件系统会自动处理细节问题。在 Linux 系统上, 可以通过使用 **stat** 命令, 方便地查看某个特定文件的 i 节点的内容。使用时, 只需键入 **stat**, 后面跟着文件的名称即可:

```
stat filename
```

- 以字节为单位的文件长度
- 包含该文件的设备名称
- 属主的用户标识
- 组 id
- 文件权限
- 上一次修改时间
- 上一次访问时间
- i 节点上一次修改时间
- 指向该文件的链接数
- 文件的类型(普通、目录、特殊、符号链接……)
- 分配给该文件的块数

图 25-4 i 节点(索引节点)的内容

索引节点或 i 节点包含文件的信息。这里示范的是 Unix 文件系统中 i 节点所存储的典型信息。各个文件系统之间 i 节点的准确内容可能有所不同。

文件系统将所有的 i 节点存放在一个大表中, 这个表称为 i 节点表(inode table)。在 i 节点表中, 每个 i 节点由所谓的索引号(index number)或 i 节点号(“i-number”)表示。例如, 假设一个特定的文件由 i 节点#478515 描述, 那么我们称这个文件的 i 节点号为 478515。为了显示文件的 i 节点号, 可以使用带-i 选项的 ls 命令。例如, 下述命令显示两个文件 **xyzyz** 及 **plugh*** 的 i 节点号:

```
ls -i xyzyz plugh
```

下述两条命令显示当前目录中所有文件的 i 节点号:

```
ls -i
ls -il
```

当处理目录时, 就好像目录实际包含文件一样。例如, 您可能听到有人说他的 **bin** 目录中包含有文件 **spacewar**。但是, 该目录中并不包含文件。实际上, 该目录只包含有文件的名称及文件的 i 节点号。因此, 目录的内容相当小: 只有一列名称, 每个名称对应一个 i 节点号。

下面举例说明。当在 **bin** 目录中创建文件 **spacewar** 时, 会发生什么情况呢? 首先, Unix 在硬盘上保留存放该文件的存储空间。接下来, Unix 查看 i 节点表, 查找一个空闲的 i 节点。假设 Unix 查找到的 i 节点是#478515。然后, Unix 将信息填充到属于该新文件的 i 节点中。最后, Unix 在 **bin** 目录中放入一个条目。该条目包含名称 **spacewar**, 以及一个为 478515 的 i 节点号。每当程序需要使用文件时, 程序在目录中查找文件名时将是一件简单的事情, 只需使用相应的 i 节点号查找 i 节点, 然后使用 i 节点中的信息访问文件即可。

文件名和 i 节点之间的连接称为**链接**。从概念上讲, 链接将文件名和文件本身连接起来。这就是为什么 i 节点不包含文件名(由图 25-4 可以看出来)的原因。实际上, 您稍后就会知道, 一个 i 节点可以由不止一个文件名引用。

25.21 多重链接

Unix 文件系统最出色的特征之一就是允许多重链接。换句话说, 就是一个文件可以有不止一个名称。这是怎么回事呢? 文件的唯一标识符是其 i 节点号, 而不是它的名称。因此, 毫无疑问, 多个文件名可以引用同一个 i 节点号。下面举例说明。

假设您的 home 目录是 **/home/harley**。在您的 home 目录中, 有一个子目录 **bin**。您在 **bin** 目录中创建了一个文件, 并命名为 **spacewar**。这个文件的 i 节点号是 478515。通过使用 **ln** 命令(本章后面描述), 您可以在同一个目录中创建一个命名为 **funky** 的文件, 该文件和 **spacewar** 拥有相同的 i 节点号。既然 **spacewar** 和 **funky** 拥有相同的 i 节点号, 所以从本质上讲, 它们是同一个文件的两个不同名称。

现在, 假设您切换到 home 目录中, 并创建了另一个文件 **extra**, 这个文件也拥有相同的 i 节点号。然后您又切换到一个朋友的 home 目录中, 即 **/home/weedly**, 并创建了第 4 个

* 当有时间时, 可以在 Internet 上查找一下这两个名称。

文件 **myfile**，该文件也拥有相同的 **i** 节点号。此时，您依然只有一个文件——由 **i** 节点号 478515 标识，但是它有 4 个不同的名称：

```
/home/harley/bin/spacewar  
/home/harley/bin/funky  
/home/harley/extra  
/home/weedly/myfile
```

为什么您希望这样做呢？一旦习惯了链接的概念，您将会发现有充足的理由使用它们。链接的基本想法就是同一个文件可能拥有不同的含义(取决于文件使用的环境)，当您更有经验并且更熟练时将会理解这一点。例如，这样可以使不同的用户以不同的名称访问同一个文件。

但是，链接背后还有更多的含义。我希望您理解链接原理的原因在于，链接是基本文件命令操作 **cp**(复制)、**mv**(移动)、**rm**(移除)以及 **ln**(链接)的基础。如果您所做的就是记住如何使用命令，那么您永远不能真正理解发生的事情，而且也永远不会理解文件系统的使用规则。

稍后，我们将讨论这一声明的含义。在这之前，我希望您考虑一个问题。假设一个文件拥有不止一个链接，也就是说，该文件可以通过不止一个名称访问。那么哪个名称是最重要的名称呢？原始名称是否拥有特殊的含义呢？答案就是 Unix 平等地对待所有的链接。它并不关心该文件的原始名称是什么。新链接和旧链接拥有同等的重要性。

在 Unix 中，文件不由它们的名称或位置控制，而只受所有权和权限控制。

25.22 创建新链接：ln

每当创建文件时，文件系统都会自动在文件名和文件之间创建一个链接。但是，有时候可能希望为已有文件创建一个新链接。当需要这样做时，可以使用 **ln**(link, 链接)命令。该命令有两种形式。第一种形式用来为单个文件创建新链接，这时使用的语法为：

```
ln file newname
```

其中 *file* 是一个已有普通文件的名称，*newname* 是希望赋予链接的名称。

例如，假设您有一个文件 **spacewar**，而您希望在 **spacewar** 和 **funky** 之间创建一个新链接，则可以使用：

```
ln spacewar funky
```

此时，您拥有了两个文件名，每个文件名都指向同一个文件(也就是说，同一个 **i** 节点号)。一旦创建了新链接，那么新链接的功能就与原始目录条目的功能相同。

第二种使用 **ln** 的方式就是为一个或多个普通文件创建新链接，并将新链接放在指定的目录中。这种使用方式的语法为：

```
ln file... directory
```

其中 *file* 是已有普通文件的名称, *directory* 是希望放置新链接的目录的名称。

下面举例说明。您的 home 目录是 `/home/harley`。在这个目录中, 拥有两个文件 `data1` 和 `data2`。您的朋友使用的 home 目录是 `/home/weedly`。在这个目录中, 有一个子目录 `work`。这个目录的文件权限允许您的用户标识创建文件。您希望创建您自己的两个文件的链接, 并将它们放在您的朋友的目录中。使用的命令为:

```
ln /home/harley/data1 /home/harley/data2 /home/weedly/work
```

为了简化该命令, 可以使用通配符(参见第 24 章):

```
ln /home/harley/data[12] /home/weedly/work
```

另一种简化该命令的方法就是在输入 `ln` 命令之前, 回到您的 home 目录中:

```
cd; ln data[12] /home/weedly/work
```

一旦创建了这些新链接, 那么两个文件将同时在两个不同的目录中拥有名称。为了查看文件的链接数, 可以使用 `ls -l` 命令。链接的数量显示在文件权限和属主的用户标识之间。例如, 假设输入了命令:

```
ls -l music videos
```

输出为:

```
-rw----- 1 harley staff 4070 Oct 14 09:50 music
-rwx----- 2 harley staff 81920 Oct 14 09:49 videos
```

可以看出, `music` 只有 1 个链接, 而 `videos` 有 2 个链接。

25.23 基本文件命令的工作方式

在分析基本文件命令时, 重要的是从文件名和链接两方面进行理解。下面介绍基本的操作:

(1) 创建文件, 创建目录[`mkdir`]

创建新文件或目录时, Unix 会留出相应的存储空间并创建 *i* 节点。然后 Unix 在适当的目录中通过使用指定的文件名或目录名以及新 *i* 节点的 *i* 节点号置入一个新条目。

(2) 复制文件[`cp`]

复制已有文件时, Unix 用源文件的内容替换目标文件的内容。但是 *i* 节点号并不进行修改。复制不存在的文件时, Unix 首先用新文件自己的 *i* 节点号创建一个全新的文件(记住, *i* 节点号才是文件的真正标识)。然后将旧文件的内容复制到新文件中。在复制之后, 将会两个相同的文件。老文件名对应于老的 *i* 节点号, 新文件名对应于新的 *i* 节点号。

(3) 重命名文件或移动文件[`mv`]

重命名或移动文件时, Unix 改变文件名, 或者移动目录条目, 或者两者都进行, 但是保持相同的 *i* 节点号。这就是同一条命令(`mv`)既可以重命名文件, 又可以移动文件的原因。

(4) 创建链接[ln]

创建已有文件的新链接时，Unix 使用指定的文件名创建一个新的目录条目，并指向原始文件的 *i* 节点号。这样，一个文件拥有两个文件名，但是两个文件名指向相同的 *i* 节点号。

(5) 移除链接[rm、rmdir]

移除链接时，Unix 通过移除目录条目，消除文件名和 *i* 节点号之间的连接。如果文件已经没有任何链接，Unix 会删除该文件。

重要的是要理解链接的移除与文件的删除并不是一回事。如果文件还有不止一个链接，那么 Unix 不会删除文件，直到移除最后一个链接。但是，在大多数情况下，文件只有一条链接，这也是为什么在大多数时候，**rm** 和 **rmdir** 都充当删除命令的原因。

下面举一个简单的例子，描述刚才讨论的思想。假设您有一个文件 **spacewar**。您决定用 **funky** 这一名称为这个文件建立一个新链接：

```
ln spacewar funky
```

现在移除 **spacewar**：

```
rm spacewar
```

尽管第一个文件名已经不存在，但是原文件依然存在。除非最后一个链接(**funky**)被移除，否则文件本身不会被删除。

25.24 符号链接：ln -s

刚才讨论的链接类型允许我们为同一个文件指定不止一个名称。但是，这样的链接有两个限制。第一，不能为目录创建链接。第二，不能为不同文件系统中的文件创建链接。

在创建不同文件系统中的目录或文件的链接时，需要创建所谓的符号链接(symbol link 或 symlink)。这样做时，需要使用带 **-s** 选项的 **ln** 命令。符号链接包含的不是文件的 *i* 节点号，而是原文件的路径名。每当访问符号链接时，Unix 借助该路径名查找文件(从这种意义上讲，符号链接类似于 Windows 的快捷键。顺便说一下，Windows Vista 实际上支持类 Unix 符号链接)。

当使用 **ls -l** 显示符号链接文件的长列表时，有两点需要注意。第一，文件类型指示符(输出最左边的字符)是小写字母 **l**，代表“link，链接”。第二，实际的符号链接在输出行的右边显示。下面举一个例子，在这个例子中有一个文件 **/bin/sh** 是一个指向 **/bin/bash** 文件的符号链接。输入命令：

```
ls -l /bin/sh
```

输出为：

```
lrwxrwxrwx 1 root root 4 Sep 11 2008 /bin/sh -> bash
```

可以看出，这个文件只有 4 个字节长，仅能容下实际文件的路径名(实际文件的路径名

只有4个字符长)。该文件是一个符号链接，而不是一个只有4个字符的普通文件，这一事实在该文件的*i*节点中注明。

如果希望查看文件本身的长列表，则必须指定实际名称：

```
ls -l /bin/bash
```

在这个例子中，输出为：

```
-rwxr-xr-x 1 root root 720888 Feb 10 2008 /bin/bash
```

可以看出，该特定文件共有720888字节。从文件名可以猜测出，该文件存放的是Bash shell程序。

为了区分两种类型的链接，一般将常规的链接称为**硬链接**(hard link)，而将符号链接称为**软链接**(soft link)。当只使用“链接”本身时，所指的是硬链接。

正如前面讨论的，为了显示某一文件硬链接的数量，可以使用**ls -l**命令。但是，没有办法显示某一文件软链接(符号链接)的数量。这是因为文件系统本身也不知道存在多少这样的链接。

问题：如果某一文件存在符号链接，那么在删除这个文件时，会发生什么情况？

答案：符号链接不会被删除。实际上，在使用**ls**命令时，仍然会显示符号链接。但是，如果试图使用该链接，则会显示错误消息。

25.25 目录使用符号链接

在第24章中，我们讨论了如何使用内置命令**cd**改变工作目录，以及使用**pwd**命令显示工作目录的名称。这就出现了一个问题：当目录名称是另一个目录的符号链接时，**cd**和**pwd**命令又起什么作用呢？答案有两种。第一种，命令可以将符号链接视为一个实体，即实际目录的一个同义词，更像普通文件的硬链接。另外一种，链接只不过是真实目录的一个跳板。

对于一些shell来说，**cd**有两个选项可以控制这样的情形。**-L**(logical, 逻辑)选项告诉**cd**将符号链接视为真实的目录。**-P**(physical, 物理)选项告诉**cd**用真实目录替换符号链接。下面举例说明。

首先，在自己的home目录中，创建一个子目录**extra**。接下来，为这个目录创建一个符号链接**backups**。最后，显示两个文件的长列表：

```
cd
mkdir extra
ln -s extra backups
ls -ld extra backups
```

下面是一些样本输出：

```
lrwxrwxrwx 1 harley staff 5 Sep 8 17:52 backups -> extra
```

```
drwxrwxr-x 2 harley staff 4096 Sep 8 17:52 extra
```

通过查看每行的第一个字符，我们可以看出 **backups** 是一个链接，而 **extra** 是一个真实的目录。注意 **backups** 只有 5 个字节长，仅能包含目标目录名。但是，目录 **extra** 有 4096 字节长，即该文件系统的块大小(参见第 24 章)。

现在，考虑下述命令：

```
cd -L backups
```

该命令将把工作目录改变成 **backups**，尽管严格地讲，**backups** 并不真正存在。如果使用 **-P** 选项，又如何呢？

```
cd -P backups
```

在这个例子中，shell 用实际目录替换符号链接，工作目录变成 **extra**。当发生这种情况时，我们说 shell 遵循该链接。

默认情况下，**cd** 假定使用 **-L** 选项，因此实际中永远不需要指定它。但是，如果希望 shell 遵循符号链接，则必须指定 **-P** 选项。

pwd(print working directory, 显示工作目录)命令在显示工作目录名时同样可以应用这两个相同的选项。为了测试这一点，我们创建上述目录和符号链接，并输入下述命令中的一条(这两条命令等效)：

```
cd backups
cd -L backups
```

现在输入命令：

```
pwd -P
```

-P 选项告诉 **pwd** 遵循链接。输出类似于下述内容：

```
/home/harley/extra
```

现在输入下述命令中的一条。对于 **cd** 来说，**-L** 选项是默认的：

```
pwd
pwd -L
```

在这种情况下，**pwd** 不遵循链接，所以输出为：

```
/home/harley/backups
```

25.26 查找与 Unix 命令相关的文件：whereis

有许多情况可能需要查找一个特定的文件或者一组文件。在这种时候，有 3 个不同的程序可供使用：**whereis**、**locate** 和 **find**。在接下来的几节中，我们将轮流讨论每个程序。

whereis 程序用来查看与特定 Unix 命令相关的文件：二进制(可执行)文件、源文件和文档文件。**whereis** 不搜索整个文件系统，而只查看那些此类文件极可能存在的目录，例如 **/bin**、**/sbin**、**/etc**、**/usr/share/man** 等(参见第 23 章中有关文件系统层次结构标准的描述)。

whereis 程序的语法为：

```
whereis [-bms] command...
```

其中 *command* 是命令的名称。

假设您希望查看与 **ls** 命令相关的文件，只需输入：

```
whereis ls
```

下面是一些典型的输出(实际上，是一个很长的行)：

```
ls: /bin/ls /usr/share/man/man1p/ls.1p.gz  
/usr/share/man/man1/ls.1.gz
```

在这个例子中，我们看到 **ls** 程序本身位于路径名为 **/bin/ls** 的文件中。这相当简单。接下来的两个路径名显示两个以压缩格式存放的不同说明书页的位置：

```
/usr/share/man/man1p/ls.1p.gz  
/usr/share/man/man1/ls.1.gz
```

(**.gz** 扩展名显示文件是用 **gzip** 程序压缩的。这样的文件在显示之前可以自动地解压缩。)

上述两行显示第一个说明书页位于第 **1p** 节中，第二个说明书页位于第 **1** 节中。通过使用 **man** 命令(参见第 9 章)可以显示任一个说明书页。因为第 **1** 节是默认的，所以不必指定它。但是，其他节必须指定，例如 **1p**。因此，用来显示这两个说明书页的命令分别为：

```
man ls  
man 1p ls
```

本例描述了我希望讨论的 **whereis** 程序的一项功能：它有时候会查找到一些您甚至都不知道其存在的说明书页。例如，在生成上述输出的系统中，**ls** 程序就有两种不同的说明书页。实际上，它们大不相同，所以这两个说明书页都值得阅读。但是，如果没有使用 **whereis** 程序进行查看，那么我们就不知道有第二个说明书页存在。

如果希望限制 **whereis** 的输出，那么可以使用 **whereis** 提供的几种选项。为了只显示可执行文件的路径名，可以使用 **-b**(binary, 二进制)选项；对于联机手册中的文件，可以使用 **-m** 选项；对于源文件，可以使用 **-s** 选项。下面举一个例子。下述命令显示 10 个不同程序的可执行文件的路径名：

```
whereis -b chmod cp id ln ls mv rm shred stat touch
```

在自己的系统上试一试这条命令，看看获得什么结果。

25.27 通过搜索数据库查看文件: locate

有两种不同的 Unix 程序提供了通用的“文件查找”服务: **locate** 和 **find**。我希望大家学会如何使用这两个程序。**find** 程序比较古老, 而且也比较难以使用。但是, 它的功能非常强大, 而且每个 Unix 和 Linux 系统都提供有这个程序。**locate** 程序较新, 且易于使用, 但是功能没有 **find** 那么强大。此外, 尽管大多数 Linux 和 FreeBSD 系统提供有这个程序, 但是并不是所有的 Unix 系统都提供这个程序(例如 Solaris)。在本节中, 我们将讨论 **locate**。在下述几节中, 我们将讨论 **find**。

locate 程序的任务就是搜索一个特殊的数据库(该数据库中包含所有可公共访问的文件的名称), 查找所有包含特定模式的路径名。该数据库自动维护, 并定期更新。**locate** 程序的语法为:

```
locate [-bcirS] pattern...
```

其中 *pattern* 是在路径名中查找的模式。

下面举一个简单的例子。您希望查找所有路径名中包含字符串“test”的文件。因为这类文件可能有许多, 所以最好将该命令的输出管道传送给 **less**(参见第 12 章), 从而每次只显示一屏输出:

```
locate test | less
```

除了普通字符, 如果使用 **-r** 选项的话, 还可以使用正则表达式。在正式表达式中, 可以分别使用 **^** 和 **\$** 表示路径名的开头和结尾(有关正则表达式的信息, 请参见第 20 章)。

下面举一个有趣的例子。假设您希望在系统中搜索照片, 一种方法就是使用扩展名 **.jpg** 或 **.png** 查找文件, 并将路径名保存在一个文件中, 然后在空闲时浏览这个文件。所使用的命令为:

```
locate -r '.jpg$' > photos  
locate -r '.png$' >> photos
```

第一条命令将输出重定向到文件 **photos** 中, 第二条命令将输出追加到同一个文件中(有关输出重定向的讨论, 请参见第 15 章)。

因为 **locate** 获得的结果通常要比希望的多, 所以要以某种方式对输出进行处理。其中最强大的组合之一就是将 **locate** 的输出管道传送给 **grep**。例如, 假设您使用的是一个新系统, 而您希望查找 Unix 的字典文件(参见第 20 章)。这个文件极有可能包含字母“dict”以及字母“words”。为了查看所有这样的文件, 可以使用 **locate** 查找所有包含“dict”的文件。然后使用 **grep** 搜索 **locate** 的输出, 查找包含“words”的行。所使用的命令为:

```
locate dict | grep words
```

在自己的系统上试一试这条命令, 看看获得什么样的结果。

为了修改 **locate** 的操作, 还可以使用几个选项。第一个选项是 **-c(count, 统计)** 选项, 该选项显示匹配文件的总数, 而不显示实际的文件名。例如, 如果希望查找系统上有多少

个 JPG 文件，可以使用命令：

```
locate -cr '*.jpg$'
```

接下来的一个选项是 **-i**(ignorecase, 忽略大小写)。该选项告诉 **locate** 将大写字母和小写字母同等对待。例如，如果希望搜索所有路径名中包含有“x11”或“X11”的文件，则可以使用下述两条命令中的任意一条命令：

```
locate x11 X11
locate -i x11
```

如果希望，还可以组合使用 **-i** 和 **-r** 选项，从而使用对大小写不敏感的正则表达式。例如，下述命令示范了如何搜索所有路径名以“/usr”开头，以“x11”或“X11”结尾的文件：

```
locate -ir '^/usr*x11$'
```

有时候，我们只需要匹配路径名的最后一部分，即所谓的文件名或基名(参见第 24 章)，这样将提供极大的方便。为了这样做，可以使用 **-b** 选项。例如，为了查看所有基名中包含字符串“temp”的文件，可以使用：

```
locate -b temp
```

为了查看所有文件名中包含字符串“temp”的文件，可以使用：

```
locate -br '^temp$'
```

最后，为了显示系统上 **locate** 数据库的信息，可以使用 **-S**(statistics, 统计)选项：

```
locate -S
```

可以看出，**locate** 非常容易使用。只需告诉自己希望的内容，**locate** 就会查找它。但是，**locate** 有一个缺点，我希望大家理解。

前面提到过 **locate** 使用一个特殊的数据库，该数据库中包含系统中所有公共可用的文件的路径名。在运行良好的系统中，该数据库会定期自动更新。但是，当您或其他人创建新文件时，新文件并不会立即出现在数据库中，要直至下一次数据库更新时才会出现在数据库中。为了避免这一限制，可以使用 **find** 程序(在下一节中讨论)，因为 **find** 实际上搜索整个目录树。

25.28 通过搜索目录树查找文件：find

到目前为止，我们已经讨论了两种不同的文件查找工具：**whereis** 和 **locate**。这两个程序都比较快速，而且易于使用，大多数时候，在需要搜索文件时，它们应该是首选程序。

但是，这两个程序存在一些限制。**whereis** 程序只搜索与特定程序关联的文件(可执行文件、源文件、文档文件)。**locate** 程序实际上并不执行搜索。它只简单地在数据库(数据库

中包含系统上所有可公共访问的文件的路径名)查找匹配的模式。当需要进行完全搜索时,可以使用的程序是 **find**。

find 程序是这 3 个程序中最古老、最复杂的程序。实际上, **find** 是最复杂的 Unix 工具之一。但是, 相对于其他程序, **find** 程序有 3 个重要的优点。第一, 它的功能非常强大, 可以根据大量不同的标准搜索任何文件, 并且可以在任何位置进行搜索。第二, 一旦 **find** 完成搜索, **find** 可以以几种不同的方式处理结果。最后, 与 **locate** 不同, 所有的 Unix 和 Linux 系统都提供有 **find** 程序, 所以可以在遇到的任何系统上使用这一程序。

find 程序的完整语法相当复杂。实际上, 我也不会向您示范这些。作为替代, 我们首先大体地介绍一下它的语法, 然后再逐步深入。

find 的一般思想就是搜索一个或多个目录树, 根据指定的测试条件, 查找满足特定标准的文件。一旦搜索完成, **find** 将对查找到的文件执行某种动作。动作可以简单得就如文件名的显示。操作还可以非常复杂: **find** 可以删除文件、显示文件的信息, 或者将文件传递给另一条命令, 进行进一步处理。

为了运行 **find**, 需要指定 3 件事情(按下述顺序): 目录路径、测试和动作。**find** 程序的一般语法为:

```
find path... test... action...
```

一旦输入该命令, **find** 就会遵循一个 3 个步骤的处理过程。

(1) 路径: **find** 所做的第一件事就是查看每个路径, 检查这些路径所表示的整个目录树, 包括所有的子目录。

(2) 测试: 对于遇到的每个文件, **find** 应用指定的测试条件。这里的目标就是创建一个满足指定标准的所有文件的列表。

(3) 动作: 一旦搜索完成, **find** 就对列表中的每个文件执行指定的操作。

考虑下述简单的例子:

```
find /home/harley -name important -print
```

现在我们还不深入讨论细节问题, 先将这条命令分成下述几部分:

路径: **/home/harley**

测试: **-name important**

动作: **-print**

在这条命令中, 我们向 **find** 指派了下述指示。

(1) 路径: 从 **/home/harley** 开始, 搜索所有的文件和子目录。

(2) 测试: 对于每个文件, 应用测试 **-name important**(该测试的含义就是查找名为 **important** 的文件)。

(3) 动作: 对于每个通过测试的文件, 执行动作 **-print**(也就是显示路径名)。

那么上述命令是干什么的呢? 该命令显示 **/home/harley** 目录树中所有命名为 **important** 的文件的路径名。

花一点时间考虑一下刚才我们如何分析上述命令。您将发现任何 **find** 命令——不论有

多复杂——都可以采用相同的方式分析，即将命令分解成 3 部分：路径、测试、动作。当需要构建复杂的 **find** 命令时，可以通过逐个考虑这 3 部分创建命令。

25.29 find 命令：路径

正如前面所讨论的，**find** 程序的一般格式为：

```
find path... test... action...
```

可以看出，每条 **find** 命令的开头由一个或多个路径构成。这些路径说明 **find** 从何处开始进行搜索。从下述例子中可以看出，路径的指定相当直接(当阅读这些例子时，要意识到这些例子没有指定任何测试或动作。我们稍后再讨论这些话题)。大多数时候，只需使用一个路径。首先，我们示范一个简单的例子：

```
find backups
```

在这个例子中，告诉 **find** 从 **backups** 目录开始，搜索该目录的所有子孙，包括文件和子目录。可以看出，我们使用了一个相对路径名。实际上也可以使用绝对路径名，一个.(点号)代表的是工作目录，一个~(波浪号)代表的是 home 目录。下面再示范几条不完整的命令：

```
find /usr/bin  
find /  
find .  
find ~  
find ~weedly
```

第一个例子告诉 **find** 从目录 **/usr/bin** 开始搜索。第二个例子从根目录开始搜索(实际上，这告诉 **find** 搜索整个文件系统)。下一个例子从工作目录开始搜索。第四个例子从 home 开始搜索。最后一个例子从用户标识 **weedly** 的 home 目录开始搜索(有关如何指定路径名的全部讨论，请参见第 24 章)。

如果希望，可以为 **find** 指定不止一个搜索路径，例如：

```
find /bin /sbin /usr/bin ~harley/bin
```

在这个例子中，**find** 将搜索 4 个独立的目录树，而搜索结果将作为一个长列表一起被处理。

25.30 find 命令：测试

我们使用 **find** 程序搜索一个或多个目录树，查找满足指定标准的文件，然后对查找到的文件执行特定的动作。为了定义标准，我们可以指定一个或多个测试。**find** 命令的一般格式为：

```
find path... test... action...
```

到目前为止，**find** 的学习相当容易。从这里开始，**find** 命令将变得复杂。在上一节中，我们讨论了如何指定路径。在本节中，将讨论如何使用各种测试指定希望处理哪些文件。测试有许多，有简单的，也有晦涩难懂的。出于参考目的，图 25-5 总结了最重要的测试。

文件名	
-name <i>pattern</i>	包含 <i>pattern</i> 的文件名
-iname <i>pattern</i>	包含 <i>pattern</i> 的文件名(不区分大小写)
文件特征	
-type [<i>df</i>]	文件类型： d =目录， f =普通文件
-perm <i>mode</i>	设置为 <i>mode</i> 的文件权限
-user <i>userid</i>	属主为 <i>userid</i>
-group <i>groupid</i>	组为 <i>groupid</i>
-size [<i>+</i>] <i>n</i> [<i>cbkMG</i>]	大小为 <i>n</i> [字符(字节)、块、千字节、兆字节、吉字节]
-empty	空文件(大小=0)
访问时间、修改时间	
-amin [<i>+</i>] <i>n</i>	<i>n</i> 分钟之前访问
-anewer <i>file</i>	<i>file</i> 文件之后访问
-atime [<i>+</i>] <i>n</i>	<i>n</i> 天之前访问
-cmin [<i>+</i>] <i>n</i>	<i>n</i> 分钟之前状态改变
-cnewer <i>file</i>	<i>file</i> 文件之后状态改变
-ctime [<i>+</i>] <i>n</i>	<i>n</i> 天之前状态改变
-mmin [<i>+</i>] <i>n</i>	<i>n</i> 分钟之前修改
-mtime [<i>+</i>] <i>n</i>	<i>n</i> 天之前修改
-newer <i>file</i>	<i>file</i> 文件之后修改

图 25-5 find 程序：测试

find 程序搜索目录树，根据各种测试，查找满足指定标准的文件。详情请参见正文。

find 是一个非常古老的程序，而且各个系统之间的基本测试是相同的。但是，较新版本的 **find** 程序支持一些并不是在所有系统上都可用的测试。图 25-5 列举了 GNU 版本的 **find** 程序可能使用的测试，GNU 版本的程序随 Linux(参见第 2 章)一起提供。如果您使用的是 一种不同类型的 Unix，那么您可以使用所有基本的测试，但是有可能无法使用一些较深奥的测试。为了查看自己版本的 **find** 程序支持的测试列表，可以查看自己系统上的说明书页 (**man find**)。

由图 25-5 可以看出，**find** 拥有许多不同的测试。最终，当需要时可以学习如何使用它们。现在，我的目的就是确信您已经理解了最重要的测试。到目前为止，两个最重要的测试为 **-type** 和 **-name**，因此我们先学习这两个测试。

-type 测试控制 **find** 查找哪些文件类型。相应的语法为 **-type** 的后面跟一个单字母的标识。大多数时候, 或者使用 **f** 代表普通文件, 或者使用 **d** 代表目录。如果需要, 还可以使用 **b**(块设备)、**c**(字符设备)、**p**(命名管道)或 **l**(符号链接)。下面举一些例子(注意: 在这些例子中, 使用的动作是 **-print**, 该动作只是简单地显示搜索的结果。我们将在本章后面讨论其他更复杂的动作)。

```
find /etc -type f -print
find /etc -type d -print
find /etc -print
```

所有 3 条命令都从 **/etc** 目录开始执行搜索。其中, 第一条命令只搜索普通文件; 第二条命令只搜索目录; 第三条命令搜索任何类型的文件。

-name 测试告诉 **find** 查找其文件名匹配指定模式的文件。如果希望, 还可以使用标准的通配符 *****、**?** 和 **[]**(参见第 24 章)。但是, 如果使用通配符, 则必须将它们引用起来, 从而在将它们传递给 **find** 时不会被 shell 解释。下面举一些例子。以下所有 3 条命令都从工作目录(**.**)开始搜索, 并且只搜索普通文件(**-type f**):

```
find . -type f -name important -print
find . -type f -name '*.c' -print
find . -type f -name 'data[123]' -print
```

第一条命令搜索名为 **important** 的文件。第二条命令搜索扩展名为 **.c** 的文件名, 也就是 C 源文件。第三条命令只搜索名为 **data1**、**data2** 或 **data3** 的文件。

提示

人们在使用 **find** 命令时最常见的错误就是使用 **-name** 时忘记引用通配符。如果不引用通配符, 那么 shell 将解释通配符, 从而产生错误。考虑下述两条命令:

```
find . -type f -name '*.c' -print
find . -type f -name *.c -print
```

第一条命令工作正常。但是, 第二条命令工作不正常, 因为 shell 将把表达式 ***.c** 解释成一组实际文件名, 从而产生语法错误。系统将显示一个类似于下述内容的错误消息:

```
find: paths must precede expression
```

和大多数 Unix 选项一样, **-name** 测试也是对大小写敏感的, 也就是说, 该选项区分大写字母和小写字母。为了忽略大小写字母之间的区别, 可以使用 **-iname** 选项。例如, 考虑下述两个例子。两条命令都从 **/usr** 开始搜索, 并且只查找目录:

```
find /usr -type d -name bin -print
find /usr -type d -iname bin -print
```

第一条命令只查找命名为 **bin** 的目录。第二条命令使用了 **-iname** 选项, 这意味着它将搜索与 **bin**、**Bin**、**BIN** 等匹配的目录。

除了名称, 还可以基于其他特征选择文件。我们已经知道了 **-type** 测试可以选择特定的

文件类型，通常 **f** 表示普通文件，**d** 表示目录。另外还可以使用 **-perm** 搜索具体模式的文件，使用 **-user** 或 **-group** 搜索由具体用户标识或组标识拥有的文件。考虑下述 3 个例子，这 3 个例子从 **home** 目录开始搜索：

```
find ~ -type d -perm 700 -print
find ~ -type f -user harley -print
find ~ -type f -group staff -print
```

第一条命令搜索文件模式为 **700** 的目录(我们已经在本章前面讨论了权限和模式)。第二条命令搜索由用户标识 **harley** 拥有的普通文件。最后一条命令搜索由组标识 **staff** 拥有的普通文件。

另外还可以根据文件的大小搜索文件，即使用 **-size** 选项，后面跟一个具体的值。这种形式的基本格式就是一个数字，后面跟一个单字母的缩写。缩写包括：表示字符的 **c**(也就是字节)、表示 512 字节块的 **b**、表示千字节的 **k**、表示兆字节的 **M** 和表示吉字节的 **G**。下面举两个例子，两个例子都从 **home** 目录开始搜索，并且搜索普通文件：

```
find ~ -type f -size 1b -print
find ~ -type f -size 100c -print
```

第一条命令搜索大小正好是一块的文件。因为这是文件的最小尺寸，所以该命令实际上搜索所有的小文件。第二条命令搜索正好包含 100 字节内容的文件。

在继续之前，我希望花一点时间解释一个重要的观点。当以块大小、千字节、兆字节或吉字节测量文件的大小时，**find** 假定所讨论的是磁盘空间。这就是为什么 **-size 1b** 查找所有的小文件的原因。正如第 24 章中所讨论的，磁盘的最小分配单元是 1 个块。

当以字节为单位测试文件大小时，**find** 假定讨论的是文件的内容，而不是使用的磁盘空间。这就是为什么 **-size 100c** 查找正好包含 100 字节数据的文件的原因。实际上，一个包含 100 字节数据的文件可以同时被 **-size 100c** 和 **-size 1b** 搜索到。

无论何时，当使用 **-size** 时，可以在数字前面加一个 **-**(减号)，或者一个 **+**(加号)，这两个符号分别表示“小于”和“大小”(这是测试所使用的所有数字的一般规则)。例如，下述第一条命令查找所有文件大小小于 10 千字节的个人文件。第二条命令查找所有文件大小大于 1 兆字节的文件：

```
find ~ -type f -size -10k -print
find ~ -type f -size +1M -print
```

最后的测试类型允许基于文件的访问时间和修改时间搜索文件。这些测试如图 25-5 所示，因此我们不再详细地讨论每个测试。我们只是举几个例子。假定您希望查找所有之前 30 分钟内修改的文件，可以使用 **-mmin** 测试，并指定值 **-30**，例如：

```
find ~ -mmin -30 -print
```

假设您希望查找所有 180 天来没有使用过的文件，则应该使用 **-atime** 测试，并指定值 **+180**：

```
find ~ -atime +180 -print
```

最后，查找所有之前 10 分钟内修改过的文件，可以使用：

```
find ~ -cmin -10 -print
```

25.31 find 命令：使用！运算符对测试求反

当需要时，可以通过在测试前面加一个！(感叹号)运算符对测试求反。这样做时，只需在测试前面键入一个！。当使用！时，必须遵循下述两条规则。第一，必须在！号的两边各留一个空格，这样才可以正确地解析！号。第二，必须引用！，这样！才可以传递给 **find**，而不会被 shell 解释(每当希望向程序传递元字符时，这是一个很好的习惯)。

作为示例，考虑下述命令，该命令搜索 home 目录，并显示所有扩展名为 **.jpg** 的普通文件：

```
find ~ -type f -name '*.jpg' -print
```

相反，假定我们希望显示那些扩展名不是 **.jpg** 的文件名。我们所需做的全部就是使用！运算符对测试求反：

```
find ~ -type f \! -name '*.jpg' -print
```

请注意，这里使用了一个反斜线引用！运算符。如果您喜欢，也可以使用单引号(有关引用的更多信息，请参见第 13 章)。

```
find ~ -type f '!' -name '*.jpg' -print
```

当必要时，还可以对多个测试求反。这种情况下，只需确保每个测试都有自己的！运算符即可。例如，假定您希望查看既不是普通文件也不是目录的文件。也就是说，您希望查看自己是否拥有符号链接、命名管道、特殊文件等。使用的命令为：

```
find ~ \! -type f \! -type d -print
```

25.32 find 命令：处理文件权限错误消息

find 程序在搜索大的目录树时是一个伟大的工具，特别是从/(根目录)开始搜索时——这样可以搜索整个文件系统。但是，当搜索自己 home 目录之外的区域时，将会发现一些目录和文件都是禁止入内的，这是因为您的用户标识没有访问它们的权限。每次发生这种情况，**find** 都将显示一个错误消息。

例如，下述命令搜索整个文件系统，查找目录名 **bin**：

```
find / -type d -name bin -print
```

当运行这条命令时，或许会看到许多与下述消息类似的错误消息：

```
find: /etc/cron.d: Permission denied
```

在大多数情况中，没有什么理由去看这些消息，因为它们并不会有什么帮助。实际上，它们只会搞乱输出。那么，如何除去它们呢？

因为错误消息写入到标准错误(参见第 15 章)中，所以可以通过将标准错误重定向到 `/dev/null`(位桶)除去错误消息。对于 Bourne Shell 家族(Bash、Korn shell)来说，这很容易：

```
find / -type d -name bin -print 2> /dev/null
```

对于 C-Shell(C-Shell、Tcsh)家族来说，这就有点复杂，不过也可以采取如下方式完成：

```
(find / -type d -name bin -print > /dev/tty) >& /dev/null
```

细节已在第 15 章中解释过。

25.33 find 命令：动作

正如前面讨论的，我们使用 `find` 程序搜索一个或多个目录树，查找满足指定标准的文件，然后对查找到的文件执行特定的动作。该命令的一般模式为：

```
find path... test... action...
```

到目前为止，我们已经讨论了如何指定路径和测试。为了完善我们的讨论，现在开始讨论动作。动作告诉 `find` 对查找到的文件执行什么操作。出于参考目的，我在图 25-6 中总结了最重要的动作(相关的完整列表，请参见系统上的 `find` 说明书页)。

动作	
<code>-print</code>	将路径名写入到标准输出
<code>-fprint file</code>	同 <code>-print</code> ；将输出写入到 <code>file</code> 中
<code>-ls</code>	显示长目录列表
<code>-fls file</code>	同 <code>-ls</code> ；将输出写入到 <code>file</code> 中
<code>-delete</code>	删除文件
<code>-exec command {} \;</code>	执行 <code>command</code> ，{} 指示匹配的文件名
<code>-ok command {} \;</code>	同 <code>-exec</code> ，但是在运行 <code>command</code> 之前进行确认

图 25-6 find 程序：动作

`find` 程序搜索目录树，根据指定的测试，查找满足标准的文件。对于查找到的每个文件，`find` 执行指定的动作。详情请参见正文。

可以看出，动作以一个 `-` 字符开头，就像测试一样。最常使用的动作就是 `-print`，该动作告诉 `find` 显示所有 `find` 选择的文件的路径名。更精确地讲，`-print` 告诉 `find` 将路径名列表写入到标准输出(为什么命名为 `-print`？正如第 7 章讨论的，出于历史原因。这是 Unix 的

传统习惯,使用单词“**print**, 打印”代表“**display**, 显示”)。

下面举一个简单、直接的例子,该命令从工作目录开始,搜索名为 **important** 的文件:

```
find . -name important -print
```

对于大多数版本的 **find** 来说,如果不指定动作,则假定 **-print** 是默认动作。因此,下述两条命令是等价的:

```
find . -name important -print
find . -name important
```

同理,对于 GNU 版本的 **find** 来说,如果不指定路径,则假定工作目录是默认路径。因此,如果您是 Linux 用户,那么下述 3 条命令是等价的:

```
find . -name important -print
find . -name important
find -name important
```

下面示范一个更有用的例子。假设您希望在整个文件系统中搜索 MP3 格式的音乐文件。这样做时,需要使用 **find** 程序从根目录开始,搜索所有扩展名为 **.mp3** 的普通文件。因为要搜索整个文件系统,所以 **find** 会生成许多文件权限错误消息(参见上一节)。基于这一原因,您将标准错误重定向到垃圾桶。完整的命令为:

```
find / -type f -name '*.mp3' -print 2> /dev/null
```

如果 MP3 文件列表过长,那么大多数文件将滚动出屏幕的范围。如果是这种情况,则有两种选择。第一,将输出管道传送给 **less**, 从而每次显示一屏:

```
find / -type f -name '*.mp3' -print 2> /dev/null | less
```

另外一种选择就是将输出保存到文件中,以便在空闲时细读该文件。这样做时,需要使用 **-fprint** 动作取代 **-print**。语法很简单:键入 **-fprint**, 后面跟着文件的名称。例如,下述命令查找系统中所有 MP3 文件的名称,并将它们存储在文件 **musiclist** 中:

```
find / -type f -name '*.mp3' -fprint musiclist 2> /dev/null
```

-print 动作显示路径名。如果希望显示每个文件更多的信息,则可以使用 **-ls** 动作。该动作显示类似于使用 **-dils** 选项的 **ls** 命令的信息。显示信息的格式化在内部完成(**find** 并不真的运行 **ls** 程序)。所显示的内容包括:

- i 节点号
- 以块为单位的大小
- 文件权限
- 硬链接数量
- 属主
- 组
- 以字节为单位的大小

- 上一次修改时间
- 路径名

作为示例，下述命令搜索 home 目录，查找所有最近 10 分钟内修改过的文件和目录。然后该命令再使用 **ls** 显示这些文件的信息：

```
find ~ -mmin -10 -ls
```

-fls 动作同 **ls** 动作，只是 **-fls** 动作与 **-fprint** 动作相似，将输出写入到文件中。例如，下述命令查看所有最近 10 分钟内修改过的文件，并将查找到的文件的路径名写入到文件 **recent** 中：

```
find ~ -mmin -10 -fls recent
```

下一个要介绍的动作是 **-delete**，该动作非常有用。但是，该动作可以很容易产生不良后果，因此在使用时一定要特别加以小心。**-delete** 动作移除搜索中查找到的每个文件的链接。如果链接只有一个，则删除该文件(参见本章前面有关链接的讨论)。换句话说，**-delete** 动作的使用类似于 **rm** 命令。

下面举例说明。从 home 目录开始搜索，且希望移除所有扩展名为 **.backup** 的文件：

```
find ~ -name '*.backup' -delete
```

下面举一个更复杂(而且更有用)的例子。下述例子移除所有扩展名为 **.backup**，而且至少有 180 天没有访问过的文件：

```
find ~ -name '*.backup' -atime +180 -delete
```

-ls 和 **-delete** 动作的使用类似于将搜索的输出分别发送给 **ls** 和 **rm** 命令。但是，**find** 程序更具有一般性：可以使用 **-exec** 动作将搜索输出发送给任何希望的程序。相应的语法为：

```
-exec command {} \;
```

其中 *command* 是您希望的任何命令，包括选项和参数。

下面介绍它的工作方式。在 **-exec** 之后键入一条命令——可以指定任何自己希望的命令，以及选项和参数，就像在命令行上键入命令一样。在命令内，可以使用字符 **{}** 表示 **find** 查找到的路径名。为了表明命令的末尾，必须以 **;**(分号)结束命令。从上述语法可以看出，分号必须被引用。这样可以确保在将分号传递给 **find** 时，**shell** 不会解释它(引用在第 13 章中讨论)。

相对于其他特性，**-exec** 动作使 **find** 的功能更为强大，因此学习如何使用该动作非常关键。为了展现 **-exec** 动作的工作方式，下面从一个不重要的例子开始。下面的命令从 home 目录开始搜索，查找所有的目录。搜索的结果发送给 **echo** 程序，每次显示一个结果。

```
find ~ -type d -exec echo {} \;
```

对于一些 **shell** 来说，如果不引用花括号，则可能会产生问题：

```
find ~ -type d -exec echo '{}' \;
```

顺便提醒读者，如果您希望，可以用一个单引号取代反斜线引用分号：

```
find ~ -type d -exec echo {} ';'
find ~ -type d -exec echo '{}' ';;'
```

在搜索过程中，每查找到一个匹配项就会执行 **-exec** 动作。在这个例子中，如果查找到 26 个目录，那么 **find** 将执行 **echo** 命令 26 次。每次执行 **echo** 命令时，**{}** 字符都将被不同目录的路径名所取代。

这所以称上一条命令是一个不重要的例子，原因在于其实不需要将路径名发送给 **echo** 来显示路径名，一般都使用 **-print** 选项：

```
find ~ -type d -print
```

但是，**-exec** 动作的功能远不止此。我们假设，出于安全性考虑，您决定除了自己，不允许其他人访问您的目录。为了实现这一策略，您需要使用 **chmod** 将所有目录的文件权限设置为 700(参见本章前面有关文件权限的讨论)。您可以为每个目录键入一个 **chmod**，而这将需要很长的时间。另一种方法是使用 **find** 搜索每个目录，然后让 **-exec** 完成该工作：

```
find ~ -type d -exec chmod 700 {} \;
```

再举一个例子。假设您使用的是一个不支持 **ls** 或 **delete** 动作版本的 **find** 程序。那么如何实现这两个动作呢？只需使用 **-exec** 运行 **ls** 或 **rm** 即可，例如：

```
find ~ -name '*.backup' -exec ls -dils {} \;
find ~ -name '*.backup' -exec rm {} \;
```

-exec 还有一个变体，允许进行更多的控制以决定执行哪条命令。使用 **-ok** 取代 **-exec**，程序就会在命令执行前向您确认每条命令：

```
find ~ -type d -ok chmod 700 {} \;
```

25.34 处理查找到的文件：xargs

当使用 **find** 搜索满足给定标准的文件时，对于查找到的文件来说有两种处理方式。第一，可以使用上一节中描述的 **-exec** 动作。这允许使用任何希望的命令处理每个文件。但是，必须要理解 **-exec** 为每个文件生成一条单独的命令。例如，如果搜索到了 57 个文件，那么 **-exec** 将生成 57 条单独的命令。

对于操作一小组文件的简单命令来说，这可能没有什么问题。但是，当搜索生成大量的文件时，还有一种更好的选择。这种方法不再使用 **-exec**，而是将 **find** 的输出管道传送给一个特殊的程序，该程序专门用来高效处理这种情形。这个程序就是 **xargs**(即 X-args)，它可以运行任何使用参数指定的命令，参数通过标准输入传递给该程序。该程序的语法为：

```
xargs [-prt] [-istring] [command [argument...]]
```

其中 *command* 是希望运行的命令的名称, *string* 是占位符, *argument* 是从标准输入读取的参数。

下面从一个简单的例子开始。假设您希望创建一个所有普通文件的列表, 并显示每个文件使用的磁盘空间。您告诉 **find** 从 **home** 目录开始, 搜索普通文件。将输出管道传送给 **xargs**, 而 **xargs** 运行 **ls -s**(参见第 24 章)显示每个文件的大小。整条命令如下所示:

```
find ~ -type f | xargs ls -s
```

下面再举一个更复杂的例子, 该例子在日常生活有极大的价值。您是一名漂亮、聪明的女士, 拥有极高的品味, 而您希望给我打电话, 告诉我您有多喜欢这本书。您记得一年前, 一个我们共同的朋友向您发送了一封电子邮件, 该邮件中包含有我的姓名和电话号码, 您将它保存到一个文件中。现在您无法查看这个文件, 因为已经记不清该文件的名称, 或者该文件保存的目录。那么您如何查找该电话号码呢?

解决方法就是使用 **find** 编辑一个所有上一次修改时间超过 365 天的普通文件的列表。将输出管道传送给 **xargs**, 并使用 **grep**(参见第 19 章)搜索所有包含字符串 “Harley Hahn” 的行。下面就是使用的命令:

```
find ~ -type f -mtime +365 | xargs grep "Harley Hahn"
```

在这个例子中, 电话号码在 **home** 目录的文件 **important** 中查找到。输出为:

```
/home/linda/important: Harley Hahn (202) 456-1111
```

当需要处理 **find** 的输出时, **echo** 命令可能出奇的有用。第 12 章中讲过, **echo** 对其参数进行求值, 并写入到标准输出中。如果传递给 **echo** 一串路径名, 那么 **echo** 将输出一个包含所有名称的长行。然后就可以将这一行管道传送给另一个程序, 用于进一步处理。

作为示例, 假设您希望统计自己有多少个普通文件和目录。您使用两条单独的命令, 一条命令统计文件, 另一条命令统计目录:

```
find ~ -type f | xargs echo | wc -w
find ~ -type d | xargs echo | wc -w
```

这两条命令都使用 **find** 搜索 **home** 目录。第一条命令使用 **-type f** 只搜索普通文件, 第二条命令使用 **-type d** 只搜索目录。两条命令都将 **find** 的输出管道传送给 **xargs**, 而 **xargs** 将输出提供给 **echo**。然后 **echo** 命令的输出被管道传送给 **wc -w**(参见第 18 章), 以统计单词的数量, 也就是路径名的数量。

有时候, 您可能希望在同一条命令中使用参数多次发送 **xargs**。为了这样做, 需要使用 **-i**(insert, 插入)选项。这将允许使用 **{ }** 作为占位符, 占位符将在命令运行之前被参数取代。下面首先从一个简单的例子入手, 示范它的工作方式。

考虑下述两条命令, 这两条命令都从工作目录开始搜索普通文件:

```
find . -type f | xargs echo
find . -type f | xargs -i echo {} {}
```

在第一条命令中, **echo** 将其参数的一份副本写入到标准输出。在第二条命令中, **echo**

将其参数的两份副本写到标准输出。下面举一个更有用的例子，该例子将工作目录中的所有文件都移动到另一个目录中，并在移动时重命名文件。

```
find . -type f | xargs -i mv {} ~/backups/{}.old
```

该命令使用 **find** 在工作目录中搜索普通文件。然后将文件列表管道传送给 **xargs**，而 **xargs** 运行 **mv** 命令(本章前面解释)。**mv** 命令将每个文件移动到子目录 **backups** 中，而目录 **backups** 位于 **home** 目录中。作为移动的一部分，扩展名 **.old** 被追加到每个文件名上。

假设您的工作目录中包含 3 个普通文件：**a**、**b** 和 **c**。在上述命令运行之后，工作目录将是空的，而 **backups** 目录中将包含 **a.old**、**b.old** 和 **c.old**。

如果希望使用 **-i**，但是由于某些原因，不希望使用 **{ }**，那么可以指定自己的占位符。只需在 **-i** 之后直接键入占位符即可，例如：

```
find . -type f | xargs -iXX mv XX ~/backups/XX.old
```

当编写这样的命令时，有可能产生无法预期的问题，因为没有办法看到会发生什么事情。如果您预期会发生问题，则可以使用 **-p(prompt, 提示)** 选项。这将告诉 **xargs** 在命令生成时，显示每条命令，并且在运行命令之前请求许可。如果键入一个以 “y” 或 “Y” 开头的响应，那么 **xargs** 将运行命令。如果键入了其他任何回答——例如 **<Return>** 键，那么 **xargs** 将忽略命令。

下面举例说明，您可以自己试一试。使用 **touch**(本章前面解释过)在工作目录中创建几个新文件：

```
touch a b c d e
```

现在输入下述命令在这些文件名的末尾都添加扩展名 **.junk**。使用 **-p** 选项，**xargs** 将为每个文件请求许可：

```
find . -name '[abcde]' | xargs -i -p mv {} {}.junk
```

如果希望查看生成了哪些命令，但是不需要请求许可，则可以使用 **-t** 选项。这将导致 **xargs** 在运行过程中显示每条命令。您可以认为 **-t** 代表着 “tell me what you are doing, 告诉我您正在做什么” 的含义：

```
find . -name '[abcde]' | xargs -i -t mv {} {}.junk
```

重点：一定要确保不要将 **-i** 选项和其他选项一起使用。例如，下述命令就不能正常工作，因为 **xargs** 将认为 **-i** 之后的 **p** 是一个占位符，而不是一个选项：

```
find . -name '[abcde]' | xargs -ip mv {} {}.junk
```

我希望大家学习的最后一个选项是 **-r**。默认情况下，**xargs** 运行指定的命令至少一次。**-r** 选项告诉 **xargs** 如果没有输入参数，则不运行命令。例如，下述命令搜索工作目录，显示空文件的长列表：

```
find . -empty | xargs ls -l
```

但是，假定当前工作目录中没有空文件，那么 **ls -l** 命令就没有任何参数。这将生成一个有关整个目录的长列表，而这并不我们所希望的。解决方法就是使用 **-r** 选项：

```
find . -empty | xargs -r ls -l
```

现在如果有参数的话，**xargs** 才会运行 **ls** 命令。

可以看出，通过将 **find** 和 **xargs** 连接在一起可以方便地构建强大的工具。但是，我并不希望给您留下这样一个印象，即 **xargs** 只和 **find** 一起使用。实际上，**xargs** 可以和任意程序一起使用，只要这些程序可以向 **xargs** 提供作为参数使用的字符串即可。下面举几个例子说明这一点。

假设您正在编写 **shell** 脚本，而且希望使用 **whoami**(参见第 8 章)显示当前的用户标识，同时使用 **date**(参见第 8 章)显示时间和日期。为了使输出结果更漂亮，您希望所有的输出都显示在一行上：

```
(whoami; date) | xargs
```

第二个例子，假设您有一个文件 **filename**，该文件包含许多文件的名称。您希望使用 **cat**(参见第 16 章)将这些文件中的全部数据组合在一起，并将输出保存到文件 **master** 中：

```
xargs cat < filenames > master
```

最后一个例子，假设您希望将当前目录中所有的 C 源文件移动到子目录 **archive** 中。首先，如果这个子目录不存在，则创建这个子目录：

```
mkdir archive
```

现在使用 **ls** 和 **xargs** 列举并移动所有扩展名为 **.c** 的文件。在这个例子中，我使用了 **-t** 选项，因此 **xargs** 在执行过程中将显示每条命令：

```
0p7 0p7
```

25.35 练习

1. 复习题

1. 创建全新的空文件时可以使用哪条命令？为什么该命令极少使用？列举 3 种文件自动创建的情况。
2. 查看下述字符串，确定它们是不是好的文件名。如果不是，解释原因。

data-backup-02	Data Backup 02
data_backup_02	Data;Backup,02
DataBackup02	databackup20
DATABACKUP02	data/backup/20

3. 什么是文件权限？文件权限的两个主要应用是什么？使用哪个程序设置或修改文件权限？3 种类型的文件权限各是什么？对于每种类型，解释它们应用普通文件和目录时各自的含义。

4. 什么是链接？什么是符号链接？什么是硬链接？什么是软链接？如何创建链接？如何创建符号链接？

5. 使用哪 3 个程序查找一个文件或一组文件？各程序分别在什么时候应用？

2. 应用题

1. 在 **home** 目录中，创建一个 **temp** 目录，并切换到这个目录。在这个目录中，创建两个子目录 **days** 和 **months**。在每个目录中，创建两个文件 **file1** 和 **file2**。提示：使用子 **shell**（参见第 15 章）切换到工作目录并创建文件。

一旦创建了所有的文件，使用 **tree** 程序（第 24 章）显示一幅包含目录和文件的目录树框图。如果自己的系统没有 **tree** 程序，则可以使用 **ls -R** 命令。

2. 继续上面的练习：

在 **days** 目录中，将文件 **file1** 重命名为 **Monday**，将文件 **file2** 重命名为 **Tuesday**。然后将 **monday** 复制到 **friday**。在 **months** 目录中，将文件 **file1** 重命名为 **december**，将文件 **file2** 重命名为 **july**。

回到目录 **temp**，使用 **tree** 显示另一幅有关目录树的框图。如果自己的系统没有 **tree** 程序，则可以使用 **ls -R** 命令。

由 **december** 创建一个链接 **april**，再创建一个符号链接 **may**。显示 **months** 目录的长列表。您发现了什么情况？

接下来，清除磁盘上的无用信息，使用一条命令删除 **temp** 目录及其所有子目录，以及这些目录中的所有文件。最后再使用一条命令确定目录 **temp** 已经不存在。

3. 使用文件编辑器创建文件 **green**。在这个文件中，输入下面一行内容：

```
I am a smart person.
```

保存文件并退出编辑器。

为文件 **green** 创建一个链接，并命名为 **blue**。显示 **green** 和 **blue** 的长列表，并注意它们的修改时间。

等待 3 分钟，然后使用文件编辑器编辑文件 **green**，将原来的文本修改成：

```
I am a very smart person.
```

保存文件并退出编辑器。

显示 **green** 和 **blue** 的长列表。为什么只编辑了一个文件，而这两个文件拥有相同的(更新)修改时间？如果修改的是 **blue**，而不是 **green** 的话，又会出现什么情况呢？

4. 假设您正在建一个网站，在这个网站中，所有的 **HTML** 文件都位于您自己的 **home** 目录下的 **httpdocs** 目录的子目录中。使用管道线查找所有扩展名为 **.html** 的文件，并将查找到的文件的权限按如下方式修改：

属主：读+写

组：无

其他：读

3. 思考题

1. 大多数基于 GUI 的文件管理器都维护着一个称为“回收站”的文件夹来存放删除的文件，以便在需要时，可以恢复删除的文件。在这样的系统上，除非将文件从“回收站”中删除，否则文件并没有被永远删除。为什么基于文本的 Unix 不提供这样的服务呢？

2. 假设返回到 1976 年那个时代的机器，那时我刚刚开始使用 Unix。您找到我，要求使用我正在用的 Unix 系统。令您惊奇的是，您看到了文件、目录、子目录、工作目录、home 目录、特殊文件、链接、i 节点、权限等。您还看到了所有的标准命令：**ls**、**mkdir**、**pwd**、**cd**、**chmod**、**cp**、**rm**、**mv**、**ln** 以及 **find**。实际上，您注意到在本章中学习的几乎每个思想和工具都已经在 30 多年前提(开发)出来了。

那么，是什么样的 Unix 文件系统基本设计使 Unix 存活了如此之久，而且仍然有用？这种现象在信息处理领域是否不寻常？

3. **find** 程序是一个功能非常强大的工具，但是也非常复杂。假设有一个基于 GUI 的程序允许从大的下拉列表中选择选项，或者在表格中输入文件模式，或者从菜单中选择各种测试，等等。这样的程序要比标准的基于文本的程序更容易使用吗？基于 GUI 的 **find** 程序能不能取代基于文本的程序？



进程和作业控制

在 Unix 中，每个对象或者由文件表示，或者由进程表示。简单地讲，文件就是一个输入源或者一个输出目标，而进程就是一个正在运行的程序。文件提供对数据的访问，而进程使事情发生。

对文件和进程有一个坚实的理解非常重要。我们已经在第 23~25 章中对文件进行了详细的讨论。在本章中，我们将讨论进程，以及与作业控制相关的话题。在讨论过程中，我们将考虑几个关键问题。进程来源于何处？系统如何管理进程？如何控制自己的进程？

在阅读本章内容时，本书前面所讨论的许多内容将会以某种方式集成到一起，从而使您获得极大的满足感。一旦理解了进程的概念，知道了如何管理进程，您就能体验到 Unix 的丰富多彩，认识到 Unix 的各个部分如何交互以形成一个复杂优美的系统。

26.1 内核管理进程的方式

在第 6 章中，我们讨论了进程的思想，进程就是一个正在执行的程序。更精确地讲，进程就是一个加载到内存中准备运行的程序，再加上程序所需的数据，以及跟踪管理程序状态所需的各种信息。所有的进程都由内核管理，内核是操作系统的核心部分。其中的细节内容，大家可以想象的到，非常复杂，因此这里仅大致介绍一下。

当进程创建时，内核赋予其一个唯一的标识号，这个标识号称为进程 ID(process id)或 PID(发音为 3 个单独的字母“P-I-D”)。为了跟踪管理系统中的所有进程，内核维护了一个进程表(process table)。按照 PID 索引，每个进程在进程表中有一个条目。除了 PID，进程表中的每个条目还包含有描述及管理进程所需的信息。

这种布局看起来熟悉吗？应该熟悉，因为它与我们在第 25 章中讨论的 i 节点号和 i 节点相似。前面讲过，每个文件都拥有一个唯一的标识号，称为 i 节点号，i 节点号用作 i 节点表的索引。每个 i 节点包含有描述及管理特定文件所需的信息。因此，进程表与 i 节点表相似。与此类似，进程 ID 对应于 i 节点号，而进程表中的条目对应于 i 节点。

小型的 Unix 系统很容易同时运行 100 多个进程。当然，有一些进程是由用户运行的程序。但是，大多数进程是自动启动的，在后台执行任务。在大型系统中，可能有数百个，甚至数千个进程，这些进程都需要共享系统的资源：处理器、内存、I/O 设备、网络连接

等。为了管理这样一个复杂的工作负荷，内核提供了一个复杂的调度服务，有时候称其为调度器(scheduler)。

调度器一直维护着一个所有正在等待执行的进程的列表。通过使用复杂的算法，调度器每次选择一个进程，给予这个进程在一个短暂的时间间隔(称为时间片)中运行的机会(在多台处理器系统中，调度器可以一次选择多个进程)。

当讨论时间片的概念时，我们通常将处理时间称为 CPU 时间。这个术语可以追溯到很久以前——现代单芯片处理器出现之前，那时大部分计算由中央处理单元或 CPU 执行。

一个典型的时间片通常是 10 毫秒(千分之十秒)的 CPU 时间。一旦时间片用尽，该进程将返回到调度列表，由内核启动另一个进程。通过这种方式，最终每个进程都将获得足够的 CPU 时间来完成它们的工作。尽管一个时间片按照人的标准来说并不是非常长，但是现代处理器非常快，因此 10 毫秒的时间实际上足以执行数万条指令。

每次进程结束时间片时，内核需要中断进程。但是，这必须通过这样一种方式实现，即当进程再次启动时，它能够在中断处继续执行。为了使这成为可能，内核需要保存每个被中断进程的数据。例如，内核要保存程序中下一条执行指令的位置、环境的副本等。

26.2 进程分叉到死亡

那么进程是如何创建的呢？除了一个著名的例外(本章后面讨论)之外，每个进程都是由另一个进程创建的。下面介绍该系统的工作方式。

正如第 2 章中讨论的，内核是操作系统的核心。因此，内核为进程提供基本的服务，具体而言包括：

- 内存管理(虚拟内存管理，包括分页)
- 进程管理(进程创建、终止、调度)
- 进程间通信(本地、网络)
- 输入/输出(通过设备驱动程序，即与物理设备实际通信的程序)
- 文件管理
- 安全和访问控制
- 网络访问(如 TCP/IP)

当进程需要内核执行服务时，它就使用系统调用发送请求。例如，进程使用系统调用初始化 I/O 操作。当编写程序时，系统调用的使用方法取决于所使用的编程语言。例如，在 C 程序中，使用的是标准库中的函数。Unix 系统一般有 200~300 个系统调用，对于其中的部分调用，程序员应该学习如何使用，至少应该知道那些重要的系统调用程序如何使用。

最重要的系统调用是那些用于进程控制和 I/O 的系统调用(参见图 26-1)。具体而言，用于创建和使用进程的系统调用有 **fork**、**exec**、**wait** 和 **exit**。

fork 系统调用创建当前进程的一个副本。一旦发生这种情况，我们就称原始进程为父进程(parent process)，或者更简单一些，称其为双亲(parent)。新进程是和父进程一模一样的副本进程，称为子进程(child process)，或者子女(child)。**wait** 系统调用强制进程暂停一会，直到另一个进程结束执行。**exec** 系统调用改变进程正在运行的程序。最后一个系统调

用 **exit** 用来终止进程。为了方便地讨论这些概念，我们通常将单词 **fork**、**exec**、**wait** 以及 **exit** 作为动词使用。例如，您可能会遇到：“When a process forks, it results in two identical processes.(当进程分叉时，将生成两个相同的进程)。”

系统调用	目的
fork	创建当前进程的一个副本
wait	等待另一个进程结束执行
exec	在当前进程中执行一个新程序
exit	终止当前进程
kill	向另一个进程发送一个信号
open	打开一个用于读取或写入的文件
read	从文件中读取数据
write	向文件中写入数据
close	关闭文件

图 26-1 常用的系统调用

许多重要的任务只能由内核执行。当进程需要执行这样的任务时，进程必须使用系统调用，向内核发送一个执行任务的请求。Unix/Linux 系统通常有 200~300 个不同的系统调用。最常用的系统调用是那些用于进程控制(**fork**、**wait**、**exec**、**exit** 和 **kill**)和文件 I/O(**open**、**read**、**write** 和 **close**)的系统调用。

令人惊奇的是，仅使用这 4 个基本的系统调用(有少数一些可以忽略的变体)，Unix 进程就能够协调您、shell 与您选择运行的程序之间错综复杂的交互。为了描述它的工作方式，下面考虑当在 shell 提示符处输入命令时会发生的事情。

从第 11 章可知，shell 是一个充当用户界面和脚本解释器的程序。正是 shell 才允许您输入命令，以及间接访问内核的服务。尽管 shell 非常重要，但一旦 shell 开始运行，它也只是个进程，是众多系统进程中的一个。与所有进程一样，shell 也拥有自己的 PID(进程 ID)，并且在进程表中拥有自己的条目。实际上，在任何时间都可以通过显示一个特殊 shell 变量的值——这个变量有一个奇怪的名称，即 \$ (美元符号)，来显示当前 shell 的 PID：

```
echo $$
```

(有关 shell 变量的讨论，请参见第 12 章。)

正如第 13 章中讨论的，命令有两种类型：内部命令和外部命令。内部命令或内置命令直接由 shell 解释，因此不需要创建新进程。但是，外部命令需要 shell 运行一个单独的进程。这样，每当希望运行外部命令时，shell 必须创建一个新进程。下面介绍其工作方式。

shell 所做的第一件事就是使用 **fork** 系统调用创建一个全新的进程。原始进程成为父进程，而新进程就成为子进程。一旦进程分叉成功，就发生两件事情。首先，子进程使用 **exec** 系统调用将它自身从运行 shell 的进程变成运行外部程序的进程。其次，父进程使用 **wait** 系统调用暂停，直到子进程结束执行。

最终，外部程序结束，此时子进程使用 **exit** 系统调用停止自身。每当进程永久停止时，不管是由于什么原因停止，我们都称进程死亡(die)或者终止(terminate)。实际上，从本章后面可以知道，当故意停止一个进程时，我们称之为“杀死(kill)”进程。

每当进程死亡时，进程所使用的所有资源——内存、文件等——都将被释放，从而可以被其他进程使用。此时，称杀死的进程为**僵进程(zombie)**。尽管僵进程是死的，而且已经不再是一个真正的进程，但是它仍然在进程表中保留着自己的条目。这是因为该条目包含着最近死亡的子进程的数据，而父进程可能对这些数据感兴趣。

在子进程成为僵进程之后，父进程(一直在耐心地等待子进程死亡)立即被内核唤醒。现在父进程有机会查看进程表中僵进程的条目，看看发生了什么结果。然后内核将进程表中的僵进程条目移除。

为了描述这一过程，下面考虑当输入运行 **vi** 文本编辑器的命令时所发生的事情。shell 所做的第一件事就是创建一个与自身相同的子进程。然后 shell 开始等待子进程的死亡。^{*} 同一瞬间，子进程使用 **exec** 使自身从运行 shell 的进程变成运行 **vi** 的进程。您观察到的现象就是，当输入 **vi** 命令的一瞬间，shell 提示符就被 **vi** 程序所取代。

当结束 **vi** 的使用并退出程序时，这将杀死正在运行 **vi** 的子进程，并将其变成僵进程。子进程的死亡致使内核唤醒父进程。这反过来致使僵进程从进程表中被移除。同时，原始进程又返回到其暂停点。您观察到的现象就是，在停止 **vi** 程序的一瞬间，您看到了一个新的 shell 提示符。

26.3 孤儿进程和废弃进程

您可能会问，如果父进程分叉之后，出乎意料地死亡，只剩下子进程，则会出现什么情况呢？当然，子进程继续执行，但是它已成为**孤儿**。孤儿进程仍然可以完成工作，但是当它死亡时，没有父进程被唤醒。最终结果是，死掉的子进程——现在以僵进程的形式存在——被遗忘在一边。

以前，孤儿僵进程将永远停留在进程表中，直到系统重新启动。现代的 Unix 系统，孤儿进程将自动地被 **#1** 进程，即 **init** 进程(本章后面讨论)收养。通过这种方式，每当孤儿进程死亡时，**init** 进程充当替身父进程，快速地清除僵进程。

当父进程创建子进程，但是没有等待子进程死亡时，也会发生相同的情形。然后，当子进程死亡时，子进程就成为僵进程。

幸运的是，这种现象并不常见。实际上，这种现象仅当程序有 **bug**，允许程序创建子进程而不等待子进程死亡时才可能发生。更有趣的是，如果您的一个程序偶然以这种方式创建了一个不死的僵进程，那么将没有直接的方法清除这个进程。毕竟，如何去杀死那些已经死掉的东西呢？

为了清除已经成为僵进程的废弃子进程，可以使用 **kill** 程序(本章后面描述)终止父进程。一旦父进程死亡，僵进程将成为孤儿，从而自动地被 **init** 进程收养。在适当的时候，**init** 进程将执行它的使命，行使继父的职责，清除僵进程最后的残余信息。

Unix 编程是不是很酷？

^{*} Unix 编程不适合那些心脏脆弱的人。

26.4 区分父进程和子进程

在本章前面，我解释过 `shell` 通过分叉创建子进程执行外部命令，然后由子进程运行命令，而父进程等待子进程终止。

这样，分叉结果生成两个相同的进程：原件(父进程)和副本(子进程)。但是一个进程必须等待，而另一个进程必须运行程序。如果父进程和子进程相同，那么父进程如何知道它是父进程，子进程如何知道它是子进程呢？换句话说，就是它们如何知道自己应该做什么呢？

答案就是当 `fork` 系统调用结束它的工作时，它向父进程和子进程各传递一个数值，这个数值称为返回值。子进程的返回值设置为 0，父进程的返回值设置为新创建进程的进程 ID。因此，在分叉操作完成之后，进程可以通过简单地查看返回值来识别它是父进程还是子进程。如果返回值大于 0，那么这个进程就是父进程。如果返回值为 0，那么这个进程就是子进程。

那么，当运行外部命令时会发生什么事情呢？在 `shell` 分叉之后，有两个相同的 `shell`。其中一个是父进程，另一个是子进程，但是，最初它们不知道谁是父进程，谁是子进程。为了识别哪个进程是父进程，哪个进程是子进程，每个进程都检查它从 `fork` 接收到的返回值。拥有正返回值的 `shell` 知道它是父进程，因此它使用 `wait` 系统调用使自己暂停。拥有 0 返回值的 `shell` 知道它是子进程，因此它使用 `exec` 系统调用运行外部程序(与其他技巧一样，一旦理解了进程的工作方式，进程看上去其实没有那么神秘)。

26.5 第一个进程：init

在下一节中，我们将开始讨论那些日常使用的程序以及用来控制进程的技术。但是，在这之前，我希望先偏离正题，讨论一个非常有趣的现象。如果进程通过分叉创建，那么每个子进程必须有一个父进程。这样，父进程也必须拥有一个自己的父进程，依此类推。实际上，如果追踪进程的生成过程追踪的足够远，可以得到一个结论，即在某个地方，必须有第一个进程。

这一结论是正确的。每个 Unix 系统都拥有一个进程——至少间接拥有——是系统中其他所有进程的父进程。各种不同类型的 Unix 之间细节差别可能很大，但是希望大家理解通用的思想。下面将用 Linux 描述具体的工作方式。

在第 2 章中，我们讨论了引导过程，即启动操作系统的一组复杂步骤。在引导过程的末尾，内核“手动”创建一个特殊的进程，而不是通过分叉。这个进程的 PID(进程 ID)为 0。基于下面将要解释的原因，我们将 #0 进程称为空闲进程(idle process)。

在执行一些重要的功能——例如初始化内核所需的数据结构——之后，空闲进程进行分叉，创建 #1 号进程。然后空闲进程执行一个非常简单的程序，该程序实质上是一个无穷的循环，不做任何事情(因此将这个进程命名为“空闲进程”)。这里的思想就是，每当没有进程等待执行时，调度器就运行空闲进程。当 #0 进程变成空闲进程时，它已经有效地完成了它的目的，然后消失。实际上，如果使用 `ps` 命令(稍后讨论)显示进程 #0 的状态，那么内核将否认该进程的存在。

那么进程#1呢？进程#1执行设置内核及结束引导过程所需的剩余步骤。基于这一原因，我们称它为**初始化进程**(init process)，并将该进程的实际程序命名为 **init**。具体而言，初始化进程打开系统控制台(参见第3章)，挂载根文件系统(参见第23章)，然后运行包含在文件 **/etc/inittab** 中的 **shell** 脚本。在这一过程中，**init** 多次分叉，创建运行系统所需的基本进程(例如，运行级别的设置，参见第6章)，并允许用户登录。在这一过程中，**init** 成为系统中所有其他进程的祖先。^{*}

与空闲进程(#0)不同，初始化进程(#1)永远不会停止运行。实际上，它是进程表中的第一个进程，而且它一直存在于进程表中，直到系统关闭。即使在系统启动之后，有时候仍然调用 **init** 来执行重要的动作。例如，正如我们刚才讨论的，当父进程在其子进程之前死亡时，子进程就成为孤儿。**init** 程序会自动地收养所有的孤儿，确保正确处理它们的死亡。

在本章后面，当讨论 **ps**(process status, 进程状态)命令时，我将说明如何显示进程及其父进程的进程 ID。您将看到如果追踪系统中任何进程的祖先足够远的话，最后都会到达进程#1。

26.6 前台进程和后台进程

当运行程序时，输入和输出通常与终端相连。对于基于文本的程序来说，输入来源于键盘，输出定位到显示器，这样才有意义，因为大多数程序为了完成任务需要与用户进行交互。

但是，一些程序可以自动完成，不需要终端的合作。例如，假设您希望使用程序从一个文件中读取非常多的数据，并对数据进行排序，然后将输出写入到另一个文件中。那么这个程序就不需要您的介入，它可以自动地完成工作。

正如前面讨论的，每当输入命令运行程序时，**shell** 在请求输入另一条命令之前，先要等待该程序的结束。但是，如果您使用前面描述的排序程序，那么就没有必要等待该程序的结束。您可以输入命令启动该程序，然后去运行下一条命令，让这个程序自己运行。

这样做时，只需在命令的末尾键入一个**&**(和号)字符即可。这将告诉 **shell** 您所运行的程序应该自己执行。例如，假设运行排序程序的命令如下所示：

```
sort < bigfile > results
```

如果输入这样的命令，那么 **shell** 将启动该程序，并等待直至该程序结束执行。仅当程序结束时，**shell** 才显示一个提示，告诉您它在等待输入一条新命令。但是，当在命令的末尾加上一个**&**字符时，命令的工作方式就完全不同了：

```
sort < bigfile > results &
```

在这种情况下，**shell** 不会等待程序结束。程序一启动，**shell** 就重新获得控制，并显示一个新提示。这意味着您可以输入另一条命令，而不必等待第一个程序结束。

^{*} 进程#0，即空闲进程，分叉创建进程#1，即初始化进程。因此，严格地讲，系统中所有进程的最终祖先实际上应该是进程#0。但是，一旦进程#0结束了它的任务，它就会消失。因此，我们可以说进程#1是系统中所有进程唯一现存的祖先。

实际上，如果某个进程对自己的家谱感兴趣，那么它不能将自己的根追踪到进程#1，因为系统不允许进程读取内核的源代码。

当 shell 在提示让用户输入一条新命令之前等待当前程序结束时，我们就称这样的进程为前台进程。当 shell 启动一个程序，但是又让该程序自己运行时，我们就称这样的程序是后台进程。在第一个例子中，我们在前台运行 **sort** 程序。在第二个例子中，我们通过向命令的末尾键入 **&** 字符在后台运行 **sort**。

正如第 15 章中解释的，大多数 Unix 程序都从标准输入(stdin)读取输入，将输出写入到标准输出(stdout)。错误消息则写入到标准错误(stderr)。当从 shell 提示行中运行程序时，stdin 与键盘相连，stdout 和 stderr 与显示器相连。如果希望改变这种方式，则可以在运行程序时重定向 stdin、stdout 和 stderr。

当前台运行进程时，从键盘读取数据，将结果写入到显示器没有问题。但是，当后台运行程序时，进程自己运行，用户可以输入另一条命令。那么，如果后台进程试图从标准 I/O 读取或者向标准 I/O 写入数据时，会发生什么情况呢？答案就是输入无法连接上，但是输出连接不会发生变化。

这有两方面的重要意义。第一，如果在后台运行的进程试图从 stdin 读取输入时，stdin 没有任何内容，该进程将无限期地暂停，等待输入。进程希望读取数据，所以它一直等待，直至获得读取的数据。在这样的情况下，唯一可以做的事情就是使用 **fg** 命令将进程移至前台(稍后讨论)。这将允许您与进程交互，给予进程所需的数据。

第二，如果后台运行的进程向 stdout 或 stderr 写入数据，输出将显示在显示器上。但是，由于您可能正在处理其他事情，因此输出将与正在处理的其他事情混杂在一起，从而使结果陷入混乱。

26.7 创建延迟: sleep

为了演示后台输出如何与前台输出混杂在一起，我们先做一个小试验。在这个试验中，我们准备在后台运行两条命令。其中第一条命令将创建一个延迟，第二条命令向终端写一些输出。同时(在延迟期间)，我们还在前台启动另一个程序。然后就可以看到当前台进程在工作，而后台进程将输出写到屏幕上时，会发生什么事情。

在开始之前，我希望先介绍用来创建延迟的工具。我们将使用程序 **sleep**，该程序的语法为：

```
sleep interval[s|m|h|d]
```

其中 *interval* 是延迟的时长。

sleep 的使用相当直接，只需以秒为单位指定希望延迟的时度。例如，为了暂停 5 秒钟，可以使用：

```
sleep 5
```

如果在终端输入这条命令，则看上去像没发生任何事情一样。但 5 秒钟之后，程序将结束，您将看到一个新的 shell 提示。

对于使用 GNU 实用工具(参见第 2 章)的 Linux 或其他系统来说，还可以在时间间隔之后指定一个单字母的限定符：**s** 代表秒(默认)，**m** 代表分钟；**h** 代表小时；**d** 代表天数。例如：

```
sleep 5
sleep 5s
sleep 5m
sleep 5h
sleep 5d
```

前两条命令暂停 5 秒，后面的三条命令分别暂停 5 分钟、5 小时和 5 天。

通常，我们在 shell 脚本中使用 **sleep** 创建一个明确的延迟。例如，假设程序 A 将程序 B 所需的数据写入一个文件中。您需要编写一个脚本，确保在程序 B 运行之前所需的数据文件已经存在。在脚本中，以循环的方式使用 **sleep** 创建一个延迟，如 5 分钟。那么每隔 5 分钟，该脚本就会检查这个文件是否存在。如果这个文件还不存在，那么脚本就再等 5 分钟，然后再进行尝试。最终，当检测到这个文件后，该脚本就继续执行，运行程序 B。

在命令行上，当在运行命令(我们希望进行的动作)之前需要等待特定的时间时，**sleep** 命令也非常有用。为了进行这样的试验，我希望大家快速地输入下述两条命令，即在一条命令之后立即输入另一条命令：

```
(sleep 20; cat /etc/passwd) &
vi /etc/termcap
```

第一条命令在后台运行。该命令暂停 20 秒，然后复制口令文件(参见第 11 章)的内容到终端上。第二条命令在前台运行。该命令使用 **vi** 文本编辑器(参见第 22 章)查看 Termcap 文件(参见第 7 章)。

在输入第二条命令之后，**vi** 文本编辑器启动，等待短暂的时间之后，将会看到口令文件的内容杂乱地显示在屏幕上。这时您就可以体会到当您正在处理其他事情时，后台进程将输出写在终端上所带来的痛苦。

从道理上讲，该怎么办呢？如果程序要从终端读取数据或者向终端写入数据，就不要在后台运行程序。

(在 **vi** 中，为了刷新屏幕，可以按 **^L** 键。对于这种情况，这是一条值得记住的便利命令。退出时，可以键入 **:q**，然后按 **<Return>** 键。)

如果程序不需要交互地运行，也就是说，如果程序不需要从键盘读取数据，那么该程序就适合于作为后台进程运行，或者向屏幕写数据。考虑前面的例子：

```
sort < bigfile > results &
```

在这种情况下，我们可以在后台运行程序，这是因为该程序从文件(**bigfile**)中获取数据，并且将结果写入到另一个文件(**results**)中。

更有趣的是，shell 允许在后台运行任何程序，而我们所需做的只是在命令之后添加一个 **&** 字符。但是，在后台运行程序时一定要深思熟虑。例如，不要在后台运行 **vi**、**less** 或其他类似的程序。

提示

如果不小心在后台运行了交互式程序，则可以通过使用 **kill** 命令终止该程序，本章后面将讨论 **kill** 命令。

技术提示

源程序的编译是一项可以在后台运行的主要活动。例如,假设您使用 `gcc` 编译器编辑 C 程序 `myprog.c`。只需确保将标准错误重定向到文件中,其他方面都会正常工作。下面的第一条命令适用于 Bourne shell 家族(Bash、Korn shell)。第二条命令适用于 C-Shell 家族(Tcsh、C-Shell):

```
gcc myprog.c 2> errors &
gcc myprog.c >& errors &
```

另一种常见的情况就是编译使用 makefile 文件的程序。例如,假设您下载了程序 `game`。在解压缩文件之后,可以使用 `make` 在后台编译该程序:

```
make game > makeoutput 2> makeerrors &
```

在这两种情形中,当程序结束执行之后,shell 都会显示一个消息。

26.8 作业控制

20 世纪 70 年代初,最初的 Unix shell 只提供了非常有限的进程控制功能。当用户运行程序时,生成结果的进程使用终端作为标准输入、标准输出和标准错误。除非进程结束,否则用户无法再输入其他命令。如果有必要在进程自己结束之前终止进程,那么用户可以按 `^C` 键发送 `intr` 信号,或者按 `^` 键发送 `quit` 信号(参见第 7 章)。两个信号之间唯一的区别就是 `quit` 信号生成一个供调试用的磁芯转储。

另外,用户还可以在命令末尾键入一个 `&` 字符以异步进程(asynchronous process)运行程序。异步进程有两个明确的特征。第一,默认情况下,标准输入与空文件/`dev/null` 相连。第二,因为进程自己运行,不需要用户输入,所以该进程不响应 `intr` 和 `quit` 信号。

现在,我们拥有了 GUI、终端窗口以及虚拟控制台,从而使同时运行多个程序变得非常容易。但是,在 20 世纪 70 年代,能够创建异步进程非常重要,因为它允许用户启动不需要从终端键入信息而单独运行的程序。例如,如果有一个需要进行编译的非常长的源程序,则可以使用异步进程完成该作业。一旦进程启动,终端可以继续使用,因此不必停止工作。当然,如果异步进程遇到了问题,那么也不能够使用 `^C` 键或者 `^` 键终止进程,只能使用 `kill` 命令(本章后面讨论)。

正如第 11 章中讨论的,原始的 Bourne Shell 由 Steven Bourne 于 1976 年在贝尔实验室创建。该 shell 属于 AT&T 公司 Unix 的一部分,而且该 shell 支持异步进程,直到 1978 年,在加利福尼亚大学伯克利分校的研究生 Bill Joy 创建一个全新的 shell C-Shell(参见第 11 章)之前,shell 还只支持异步进程。在 C-Shell 中, Joy 添加了一个新功能,即作业控制(Joy 还添加了其他几个重要的新功能,如别名和命令历史)。

作业控制使多个进程的同时运行成为可能:一个进程在前台运行,其他进程在后台运行。在 C-Shell 中,用户可以暂停任何进程,而且还可以在需要时重新启动进程。用户还可以在前台进程和后台进程之间切换、挂起(暂停)进程以及显示进程的状态。Joy 在 BSD(Berkeley Unix)中包含了 C-Shell,而且事实证明作业控制是该 shell 最流行的特性之一。

即便如此，AT&T 公司的 Unix 仍然没有包含作业控制，直到 4 年之后，David Korn 于 1982 年在最初的 Korn Shell 中才包含作业控制。现在，每个重要的 Unix shell 都支持作业控制。

作业控制的本质特性就是将每条输入的命令视为一个作业，该作业由一个唯一的作业号(job number，也称为作业 ID，发音为“job-I-D”)来标识。为了控制和管理作业，可以将作业 ID 和一系列命令、变量、终端设置、shell 变量以及 shell 选项一起使用。出于参考目的，图 26-2 列举了这些工具。

作业控制命令	
jobs	显示作业列表
ps	显示进程列表
fg	将作业移至前台
bg	将作业移至后台
suspend	挂起当前 shell
^Z	挂起当前前台作业
kill	向作业发送信号；默认情况下，终止作业

变量	
echo \$\$	显示当前 shell 的 PID
echo \$!	显示上一条移至后台的命令的 PID

终端设置	
stty tostop	挂起试图向终端写数据的后台作业
stty -tostop	关闭 tostop

shell 选项: Bash、Korn shell	
set -o monitor	允许作业控制
set +o nomonitor	关闭 monitor
set -o notify	当后台作业结束时立即通报
set +o nonotify	关闭 notify

shell 变量: Tcsh、C-Shell	
set listjobs	不管作业是否挂起，列举所有的作业(只适用于 Tcsh)
set listjobs long	长列表的 listjobs(只适用于 Tcsh)
set notify	当后台作业结束时立即通报
set nonotify	关闭 notify

图 26-2 作业控制：工具

作业控制是一项由 shell 支持的特性，允许同时运行多项作业，其中一个作业在前台运行，其他作业在后台运行。作业可以有选择地挂起(暂停)、重新启动、在前台和后台之间切换，以及显示状态。这样做时，需要使用一系列的命令、变量、终端设置、shell 变量和 shell 选项。

在 Bourne Shell 家族(Bash、Korn Shell)中, 当设置 **monitor** 选项时才允许作业控制。对于交互式的 shell 来说, 这是默认的, 但是通过关闭该选项(参见第 12 章)可以关闭作业控制。在 C-Shell 家族(Tcsh、C-Shell)中, 对于交互式 shell 来说, 作业控制总是处于打开状态。

人们很自然地想知道, 作业和进程之间有什么区别呢? 在很大程度上, 这两个概念是相似的, 人们通常交换着使用术语“作业”和“进程”。但严格地讲, 它们之间还是有区别的。进程是正在执行或者准备执行的程序。作业指解释整个命令行所需的全部进程。进程由内核控制, 而作业由 shell 控制。内核使用进程表记录进程, 而 shell 也采用相同的方式, 使用作业表(job table)记录作业。

作为示例, 下面假设您输入下述简单的命令, 显示时间和日期:

```
date
```

该命令生成一个单独的进程(该进程拥有自己的进程 ID), 和一个单独的作业(该作业拥有自己的作业 ID)。在作业运行时, 进程表中会有一个新的条目, 作业表中也会有一个新的条目。考虑下述更复杂的命令行。第一条命令使用一个由 4 个不同程序构成的管道线。第二条命令按顺序执行 4 个不同的程序。

```
who | cut -c 1-8 | sort | uniq -c
date; who; uptime; cal 12 2008
```

这两个命令行各自生成 4 个不同的进程, 每个程序一个进程, 每个进程都拥有自己的进程 ID。但是, 整个管道线——不管它需要多少进程——被认为是一个单独的作业, 拥有一个单独的作业 ID。在该作业运行时, 进程表中有 4 个条目, 但是作业表中只有一个条目。

在任何时候, 使用 **ps**(process status, 进程状态)命令都可以显示所有进程的列表。同样, 使用 **jobs** 命令可以显示作业的列表。我们将在本章后面对此加以详细讨论。

26.9 在后台运行作业

为了在后台运行作业, 需要在命令末键入一个 **&** 字符。例如, 下述命令在后台运行 **ls** 程序, 并且将输出重定向到文件 **temp** 中:

```
ls > temp &
```

每次在后台运行作业时, shell 都会显示作业的作业号和进程 ID。shell 从 1 开始, 自己为作业分配作业号。例如, 如果您创建了 4 个作业, 那么它们将被分别赋予作业号 1、2、3 和 4。内核分配进程 ID, 在大多数情况下, 进程 ID 是一个多位的数字。

作为示例, 下面假设您输入了上述命令。shell 的显示如下所示:

```
[1] 4003
```

这意味着作业号#1 已经被启动, 其进程 ID 为 4003。如果作业是一个由多个程序构成

的管道线, 那么看到的进程 ID 是管道线中最后一个程序的进程 ID。例如, 假设您输入了:

```
who | cut -c 1-8 | sort | uniq -c &
```

shell 的显示如下所示:

```
[2] 4354
```

这说明您已经启动了作业#2, 最后一个程序(**uniq**)的进程 ID 是 4354。

因为后台作业自己运行, 所以没有什么简单的方法来了解它们的进展。基于这一原因, 每当后台作业结束时, shell 都会发送一个短的状态消息。例如, 当第一个例子中的作业结束时, shell 将显示一个类似于下面的消息:

```
[1] Done ls > temp
```

该消息通知您作业#1 已经结束。

如果您正在等待一个特定的后台作业结束, 那么这样的通知非常重要。但是, 当您正在做某些事情时(例如, 编辑文件或者阅读说明书页), 如果 shell 随意地显示这样的状态消息, 将使您非常烦恼。基于这一原因, 当后台作业结束时, shell 不会立即通知您。shell 会一直等待, 直到要显示下一个 shell 提示时。这将防止状态消息干扰另一个程序的输出。

如果不希望等待, 则有一个设置项可用来强制 shell 在后台作业结束时立即通知您, 而不管您正在做什么。对于 Bourne shell 家族(Bash、Korn Shell)来说, 需要设置的是 **notify** 选项:

```
set -o notify
```

取消该选项时使用的命令是:

```
set +o notify
```

对于 C-Shell 家族(Tcsh、C-Shell)来说, 设置的是 **notify** 变量:

```
set notify
```

取消该变量时使用的命令是:

```
unset notify
```

有关 shell 选项和 shell 变量使用方式的讨论, 请参见第 12 章。如果希望使设置永久化, 可以在环境文件(参见第 14 章)中放入合适的命令。

26.10 挂起作业: fg

在任何时候, 每个作业都处于 3 种状态中的一种: 前台运行; 后台运行; 暂停, 等待信号恢复执行。当需要暂停前台作业时, 可以按 **^Z** 键(Ctrl-Z)。正如第 7 章中讨论的, 这将发送 **susp** 信号, 从而使进程暂停。当通过这种方式暂停进程时, 我们称将进程挂起

(suspend), 或者将进程停止(stop)。

该术语可能有点容易使人误解, 所以我们讨论一下这个术语。术语“stop, 停止”指的是临时中止。实际上, 可以看出, 一个停止的作业还可以重新启动。因此, 当按`^Z`键时, 它只是暂停作业。如果希望永久地停止进程, 则必须按`^C`键或者使用 `kill` 命令(这两种方式都将在本章后面讨论)。

当停止程序时, `shell` 就会中断程序, 并显示一个新的 `shell` 提示。这样就可以继续输入命令了。当希望恢复挂起的程序时, 可以使用 `fg` 命令将该程序移回前台。通过以这种方式使用`^Z`和`fg`, 可以挂起程序, 输入一些命令, 然后再在希望时返回原来的程序。下面给出一个典型的例子, 示范如何利用这一功能。

假设您正在使用 `vi` 文本编辑器编写一个 `shell` 脚本。在该脚本中, 您希望使用 `cal` 命令显示日历, 但是您不确定该命令的语法。于是您挂起 `vi`, 显示 `cal` 的说明书页, 查看希望的内容, 然后再返回到离开 `vi` 的位置。下面示范具体过程。首先输入下述命令运行 `vi`:

```
vi script
```

您现在编辑的文件是 `script`。假设您已经键入了几行脚本, 接下来需要查看 `cal` 程序的使用方法。为了挂起 `vi`, 您按下`^Z`键:

```
^Z
```

```
shell 暂停 vi, 并显示一个消息:
```

```
[3]+ Stopped vi script
```

在这个例子中, 该消息告诉您, `vi`, 即作业#3 已经挂起。现在您位于 `shell` 提示中。输入命令显示 `cal` 的说明书页:

```
man cal
```

浏览 `cal` 的说明书页, 然后按 `q` 键退出。您又会看到 `shell` 提示。现在您可以通过将 `vi` 移回到前台重新启动 `vi`:

```
fg
```

现在您又返回 `vi`, 并且正好位于离开 `vi` 时的位置上(当希望退出 `vi` 时, 可以键入:`q` 并按`<Return>`键)。

提示

如果您正在工作, 突然程序停止, 并看到一个类似于“Stopped”或“Suspended”的消息, 那么这意味着您可能不小心按下了`^Z`键。

当发生这种情况时, 只需输入 `fg`, 您的程序就会回来。

当挂起作业时, 进程会无限期停止。如果试图注销系统, 则可能产生问题, 因为挂起的作业还在等待完成。这种情况下的规则就是, 当注销系统时, 所有挂起的作业自动终止。在大多数情况下, 这是一种过失。因此, 如果在试图注销系统时, 还存在挂起的作业, 那

么 shell 将显示一个警告消息。下面是一些示例：

```
There are suspended jobs.
You have stopped jobs.
```

如果在试图注销系统时看到这样的消息，则可以使用 **fg** 将挂起的作业移回到前台，并正确地退出程序。如果有不止一个挂起的程序，则必须为每个挂起程序重复这一过程。这样就会防止由于不小心丢失数据。

有时候，即使有不止一个挂起的作业，您也完全确信可以注销系统。如果是这样，您只需第二次注销系统。因为 shell 已经警告过一次，所以 shell 会假定您知道自己正在做什么，这一次 shell 将允许您注销系统，而不会给出警告。但是，一定要记住，通过这种方式注销系统将终止所有挂起的程序，这些程序在下次登录时就会不处于运行状态了。

Tcsh 用户提示

当在 Tcsh 中挂起作业时，shell 只显示一个短消息 “Suspended”，而不提供其他信息。但如果设置 **listjobs** 变量的话，Tcsh 将显示所有挂起作业的列表。使用的命令为：

```
set listjobs
```

如果为 **listjobs** 赋予值 **long**，那么 Tcsh 显示的信息将是“长”列表，包括每个作业的进程 ID：

```
set listjobs=long
```

我的建议就是将这条命令放在环境文件(参见第 14 章)中，从而使该设置永久化。

26.11 挂起 shell: suspend

按下 **^Z** 键将挂起在前台运行的任何作业。但有一个进程不会挂起，这个进程就是当前的 shell。如果希望暂停当前的 shell，则需要使用 **suspend** 命令。该命令的语法为：

```
suspend [-f]
```

为什么希望挂起 shell 呢？下面举例说明。正如第 4 章中讨论的，当拥有自己的 Unix 或 Linux 系统时，自己必须进行系统管理。假设您以自己的用户标识登录，并且希望做一些要求超级用户身份的工作，因此您使用 **su** 命令(参见第 6 章)以 **root** 身份启动一个新的 shell。在做了一些超级用户的工作之后，您意识到自己需要以自己的用户标识做一些工作。停止超级用户的 shell，稍后再重新启动超级用户的 shell 并不是理想的方法，因为这样就会失去当前的工作目录、变量的改变等。作为替代，可以输入：

```
suspend
```

这将暂停当前的 shell——作为超级用户的 shell，并返回到以自己的用户标识登录的 shell。当准备好再次返回到超级用户，继续完成管理工作时，可以使用 **fg** 命令将超级用户

的 shell 移回到前台：

fg

下面再举一个例子。假设您使用 Bash 作为默认 shell，但是您希望体验一下 Tcsh 的使用。因此您输入下述命令启动一个新的 shell：

tcsh

在任何时候，您都可以使用 **suspend** 暂停 Tcsh，并返回到 Bash。稍后，可以使用 **fg** 恢复 Tcsh。

挂起 shell 的唯一限制就是，默认情况下，不允许挂起登录 shell。这样可以防止在停止主 shell 的情况下陷入危险的边缘。但是，在特定的环境中，有时也可能希望暂停登录 shell。例如，当通过 **su -** 而不是 **su** 启动超级用户 shell 时，将创建一个登录 shell(参见第 6 章)。如果希望挂起这个新 shell，则必须使用 **-f(force, 强制)** 选项：

suspend -f

这将告诉 **suspend** 暂停当前的 shell，而不管该 shell 是否登录 shell。

26.12 作业控制与多窗口

在第 6 章中，我们讨论了在自己的计算机上使用 Unix 或 Linux 时，同时运行多个程序的各种方法。首先，我们可以使用多个虚拟控制台，每个控制台支持一个完全独立的工作会话。其次，在 GUI 中，可以打开任意多个终端窗口，每个终端窗口都拥有自己 shell 的 CLI(命令行界面)。最后，一些终端窗口程序允许在同一个窗口中拥有多个标签，每个标签有自己的 shell。

拥有如此众多的灵活性，为什么还需要挂起进程以及在后台运行程序呢？为什么不让每个程序在自己的窗口中运行，不使用作业控制呢？对于这些问题有几个重要的答案。

首先，如果每次开始新任务时都需要在不同的虚拟控制台、窗口或者标签之间切换，那么工作就会非常慢。在许多情况下，简单地暂停当前的工作，输入几条命令，然后再返回到原来的任务中，这样就没有那么烦人。

其次，当使用多个窗口时，屏幕上会有许多可视元素，从而会减缓速度。此外，窗口也需要管理：移动、调整大小、图标化、最大化等。在使用作业控制时，通过减少心理以及视觉混乱，可以使生活非常简单。

第三，在短时间内使用的命令通常与特定的任务或问题相关。在这种情况下，从历史列表(参见第 13 章)中调用先前的命令比较方便。当使用分开的窗口时，某个窗口中的历史列表无法访问另一个窗口中的历史列表。

最后，有时候需要使用终端仿真器访问远程主机(参见第 3 章)，特别是对系统管理员来说。在这种情况下，只有一个 CLI 与远程主机相连。没有多个窗口或者虚拟控制台的 GUI。如果您对作业控制的使用还没有那么熟练，则每次只能运行一个程序，而这会使人

感到灰心。

通常，当需要在完全不相关的任务——特别是需要全屏的任务——之间切换时，使用多个窗口或者分开的虚拟控制台就比较合理。但是，在其他大多数情况中，作业控制会更好更快。

26.13 显示作业列表: jobs

在任何时候，可以使用 **jobs** 命令显示所有作业的列表。该命令的语法为：

```
jobs [-l]
```

在大多数情况下，只需输入该命令本身：

```
jobs
```

下面是一些样本输出，其中有 3 个挂起的作业(#1、#3、#4)和一个在后台运行的作业(#2)：

```
[1]  Stopped  vim document
[2]  Running  make game >makeoutput 2>makeerrors &
[3]- Stopped  less /etc/passwd
[4]+ Stopped  man cal
```

如果希望查看作业的进程 ID 以及作业号和命令名，则可以使用 **-l**(long listing, 长列表)选项：

```
jobs -l
```

输出如下：

```
[1]  2288 Stopped  vim document
[2]  2290 Running  make game >makeoutput 2>makeerrors &
[3]- 2291 Stopped  less /etc/passwd
[4]+ 2319 Stopped  man cal
```

注意在两个列表中，各有一个作业标记有一个+(加号)字符。这个作业称为“当前作业”。另外还有一个作业标记有一个-(减号)字符。这个作业称为“前一个作业”。

这些表示方法由管理作业的各种命令使用。如果不指定作业号，这些命令默认情况下作用于当前作业(当讨论 **fg** 和 **bg** 命令时，大家将会明白这一点)。在大多数情况中，当前作业是最近挂起的那个作业。前一个作业是队列中的下一个作业。在我们的例子中，当前作业是#4，前一个作业是#3。

如果没有挂起的作业，那么当前作业就是最近移至后台的那个作业。例如，假设您输入了 **jobs** 命令，看到如下输出：


```
[2] Running make game >makeoutput 2>makeerrors &
[6]- Running calculate data1 data2 &
[7]+ Running gcc program.c &
```

在这个例子中，没有挂起的作业。但是，后台运行的作业有3个。当前作业是#7。前一个作业是#6。

26.14 将作业移至前台：fg

当需要将作业移至前台时，可以使用 **fg** 命令。该命令的语法有3种变体：

```
fg
fg %[job]
%[job]
```

其中 *job* 表示一个特定的作业。

尽管该命令的语法看上去有点难解，但实际上相当简单。该命令最简单的形式就是输入命令本身：

```
fg
```

这将告诉 shell 重新启动当前作业，即在使用 **jobs** 命令时那个标记有+字符的作业。例如，假设您使用了 **jobs** 命令，输出为：

```
[1] 2288 Stopped vim document
[2] 2290 Running make game >makeoutput 2>makeerrors &
[3]- 2291 Stopped less /etc/passwd
[4]+ 2319 Stopped man cal
```

当前作业是#4，该作业被挂起。如果输入 **fg** 命令本身，那么该命令通过将作业#4 移至前台，重新启动作业#4。

下面假设在另一种情形中，您又输入了 **jobs** 命令，输出为：

```
[2] Running make game >makeoutput 2>makeerrors &
[6]- Running calculate data1 data2 &
[7]+ Running gcc program.c &
```

在这个例子中，当前作业是#7，该作业在后台运行。如果输入 **fg** 命令本身，该命令将把作业#7 从后台移至前台。这样将允许用户与该程序进行交互。

为了移动一个不是当前作业的作业，必须明确地标识这个作业。实现该操作的方法有若干种，图 26-3 汇总了各种方法。

作业号	含义
%%	当前作业
%+	当前作业
%-	前一个作业
%n	作业#n
%name	含有指定命令名的作业
%?name	命令中的任意位置含有 <i>name</i> 的作业

图 26-3 作业控制：指定作业

为了使用作业控制命令，必须指定一个或多个作业。作业的指定有多种不同方式：代表当前作业、代表前一个作业、使用特定的作业号或者使用命令名称的全部或部分。

大多数时候，指定作业最容易的方法就是使用一个%(百分比)字符，后面跟着作业号。例如，为了将作业#1 移至后台，可以使用：

```
fg %1
```

另外，还可以通过引用命令的名称指定作业。例如，如果希望重新启动运行命令 **make game** 的作业，则可以使用：

```
fg %make
```

实际上，只要指定命令的足够部分，使其能够从所有作业中区分出某一命令即可。如果再没有其他以字母“m”开头的命令，则可以使用：

```
fg %m
```

另外一种可选的方式是使用%?，后面跟部分命令名。例如，另一种将 **make game** 命令移至前台的方式就是使用：

```
fg %?game
```

正如前面所述，如果使用 **fg** 命令时没有指定特定的作业，那么 **fg** 将把当前作业移至前台(也就是在使用 **jobs** 命令时，输出中标记有+字符的那个作业)。另外，还可以使用%或%+指定当前作业。因此，下面 3 条命令是等价的：

```
fg
fg %
fg %+
```

同样，可以使用%-指定前一个作业：

```
fg %-
```

该命令处理的的就是在使用 **jobs** 命令时，输出中标记有-字符的那个作业。

为了方便起见,一些 shell(Bash、Tcsh、C-Shell)假定如果只是输入一个以%字符开头的命令行,那么使用的命令就是 **fg** 命令。例如,假设 2 号作业是命令 **vim document**,而且没有其他作业使用相似的名称。那么下面几条命令拥有相同的结果:

提示

为了在两个作业之间快速地切换,可以使用:

```
fg %-
```

一旦习惯了这一命令,您就会大量地使用这条命令。

```
%2
fg %2
fg %vim
fg %?docu
```

在这些命令中,shell 都将 2 号作业移至前台。

对于一些 shell 来说,还有最后一个缩写可以使用:该命令不包含其他内容,只有一个单独的字符%,这将告诉 shell 将当前作业移至前台。因此下面 4 条命令是等价的:

```
%
fg
fg %
fg %+
```

您是否注意到一些有趣的事情?如果只键入作业标记本身,那么 shell 将假定希望使用的命令是 **fg** 命令。因此,**fg** 是唯一一条命令名本身可选的命令。记住这种琐碎的有趣事情,某天,它可能帮您取胜朋友,影响他人。

提示

尽管我们的例子中都显示了几个同时挂起的作业,但是通常只需要暂停一个作业,做点其他事情,然后再返回到原作业。

在这种情况下,作业控制非常简单。为了挂起作业,只需按下[^]Z 键。为了重新启动作业,可以输入 **fg**,或者输入%(如果 shell 支持的话)。

26.15 将作业移至后台: **bg**

为了将作业移至后台,需要使用 **bg** 命令。该命令的语法为:

```
bg [%job...]
```

其中 *job* 标识一个特定的作业。

在指定作业时，需要遵循和 **fg** 命令相同的规则，特别是可以使用图 26-3 中的各种变体。例如，为了将 2 号作业移至后台，可以使用：

```
bg %2
```

如果喜欢，还可以同时将多个作业移至后台，例如：

```
bg %2 %5 %6
```

为了将当前作业移至后台，可以使用命令本身，而无需指定作业号：

```
bg
```

可以想象出，**fg** 命令的使用要比 **bg** 命令的使用更频繁。但是，在一种重要的情形下，**bg** 命令会提供极大的便利。假设您输入了一条似乎要运行好长时间的命令。如果这个程序不是交互式的，那么可以挂起这个程序，将该程序移至后台。例如，假设您希望使用 **make** 编译程序 **game**，因此您输入命令：

```
make game > makeoutput 2> makeerrors
```

在等待一会儿之后，您意识到这条命令还要运行很长时间。因为 **make** 不需要您输入内容，所以不需要从终端键入内容。您只需简单地按下 **^Z** 键挂起该作业，然后输入 **bg** 将该作业移至后台。接下来终端就可以自由使用了。

提示

当准备在后台运行程序，但是在输入命令的过程中忘了键入 **&** 字符时，**bg** 命令特别有用。如果没有键入 **&** 字符，那么作业将在前台运行。

这种情况下，只需按下 **^Z** 键挂起作业，然后使用 **bg** 命令将作业移至后台。

26.16 学习使用 ps 程序

在显示进程的信息时，可以使用 **ps**(process status, 进程状态)程序。**ps** 程序是一个非常有用的工具，可以帮助查找特定的 PID(进程 ID)，检查进程正在做什么，并且概述系统中正在发生的每件事情。然而，**ps** 拥有许多使人迷惑且不鲜明的选项，仅阅读其说明书页可能会让人很失望。

这种情形的产生有几种原因。首先，正如第 2 章中讨论的，在 20 世纪 80 年代，Unix 有两种主要分支：官方的 Unix(AT&T 公司)和非官方的 Unix(加利福尼亚大学伯克利分校)。UNIX 和 BSD 都有自己的 **ps** 版本，每个版本都有自己的选项。随着时间的推移，两种类型的 **ps** 都很出名，并且被广泛使用。

最终结果是,许多现代版本的 **ps** 同时支持两种类型的选项,即所谓的 **Unix 选项**和 **BSD 选项**。例如, **Linux** 就是这种情况。因此,对于 **Linux** 版本的 **ps** 来说,可以根据自己的爱好,或者使用 **UNIX** 选项,或者使用 **BSD** 选项。但有时候,可能还会遇到只支持一种选项,即只支持 **UNIX** 选项或者 **BSD** 选项的 **ps**。因为我们不可能知道将会使用什么样的系统,所以必须熟悉两种类型的选项。

其次, **ps** 是一个功能强大的工具,由系统管理员和高级程序员用来进行各种类型的分析。如此一来,有许多技术选项在日常使用中并不需要。然而,在阅读说明书页时,这些选项仍然存在,而这些描述可能使人迷惑。

第三,如果是使用 **GNU** 实用工具的系统——如 **Linux**(参见第 2 章),那么 **ps** 不仅支持 **UNIX** 选项和 **BSD** 选项,而且还支持一组只适用于 **GNU** 的选项。但在大多数时候,这些选项都可以忽略。

最后,更加使人容易迷惑的是,有时候将 **UNIX** 选项称为 **POSIX** 选项或者标准选项。这是因为 **UNIX** 选项是 **POSIX** 版本的 **ps** 的基础(**POSIX** 是一个大规模的计划,从 20 世纪 90 年代开始,其目标是标准化 **Unix**,参见第 11 章)。

现在,已经非常明显,为了全面学习这些内容,我们需要一个计划,下面介绍具体的计划。

尽管 **ps** 拥有许多的选项,但是日常工作仅需要少数几个选项。本书的计划就是讲授学习 **UNIX** 选项和 **BSD** 选项所需知道的最少内容。我们将忽略所有的深奥选项,包括只适用于 **GNU** 的那些选项。那么,您会不会需要其他选项呢?当然有可能需要,这时只需查看系统上的 **ps** 说明书页(**man ps**),看看有什么选项可以使用。

26.17 ps 程序: 基本技能

当需要显示进程的信息时,可以使用 **ps** 程序。正如前面所讨论的, **ps** 拥有众多的选项,这些选项可以分成 3 组: **UNIX** 选项、**BSD** 选项和仅适用于 **GNU** 的选项。后面将讲授如何使用最重要的 **UNIX** 和 **BSD** 选项,这些选项通常可以满足日常的全部需要。

对于 **ps** 选项来说,有一个有趣的传统。**UNIX** 选项通常以连字符(-)开头,而 **BSD** 选项前面没有连字符。当阅读说明书页时一定要记住这一点:如果选项前面有连字符,那么这个选项是 **UNIX** 选项;如果选项前面没有连字符,那么这个选项就是 **BSD** 选项。在我们的讨论中,将遵循这一传统。

如果您使用的 **ps** 同时支持 **UNIX** 选项和 **BSD** 选项,那么您可以使用任何一种自己喜欢的选项。实际上,有经验的用户有时候使用这一组选项,有时候使用另外一组选项,但是总是选择一组最适合解决问题的选项。不过,在此要警告大家,不要在同一条命令中混合使用两种类型的选项,这样会产生微妙的问题。

开始之前，我们先给出 `ps` 使用 UNIX 选项的基本语法：

```

ps [-aefFly] [-p pid] [-u userid]
```

以及使用 BSD 选项的基本语法：

```

ps [ajluvx] [p pid] [U userid]
```

在两种情况中，`pid` 是进程 ID，`userid` 是用户标识。

在单独介绍每个选项之前，我们将所需了解的内容摘要汇总在几个表中。图 26-4 中包含的是 `ps` 在使用 UNIX 选项时需要了解的信息。图 26-5 中包含的是 `ps` 在使用 BSD 选项时需要了解的信息。下面先花一点时间浏览这两张图。起初，它们可能有点令人费解，但是习惯之后，就会理解所有事情。

显示哪些进程？	
<code>ps</code>	与您的用户标识和终端相关的进程
<code>ps -a</code>	与任何用户标识和终端相关的进程
<code>ps -e</code>	所有进程(包括守护进程)
<code>ps -p pid</code>	与指定进程 ID <code>pid</code> 相关的进程
<code>ps -u userid</code>	与指定用户标识 <code>userid</code> 相关的进程

显示哪些数据列？	
<code>ps</code>	<code>PID TTY TIME CMD</code>
<code>ps -f</code>	<code>UID PID PPID C TTY TIME CMD</code>
<code>ps -F</code>	<code>UID PID PPID C SZ RSS STIME TTY TIME CMD</code>
<code>ps -l</code>	<code>F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD</code>
<code>ps -ly</code>	<code>S UID PID PPID C PRI NI RSS SZ WCHAN TTY TIME CMD</code>

有用的特殊组合	
<code>ps</code>	显示自己的进程
<code>ps -cf</code>	显示所有用户进程，完整输出
<code>ps -a</code>	显示所有非守护进程的进程
<code>ps -t -</code>	(仅显示所有守护进程)

图 26-4 `ps` 程序：UNIX 选项

`ps` 程序显示系统中正在运行的进程的信息。`ps` 程序有两组选项可供使用：UNIX 选项和 BSD 选项。本表列举的是一些最重要的 UNIX 选项。

显示哪些进程？	
ps	与您的用户标识和终端相关的进程
ps a	与任何用户标识和终端相关的进程
ps e	所有进程(包括守护进程)
ps p pid	与指定进程 ID <i>pid</i> 相关的进程
ps U userid	与指定用户标识 <i>userid</i> 相关的进程

显示哪些数据列？	
ps	PID TT STAT TIME COMMAND
ps j	USER PID PPID PGID SESS JOBC STAT TT TIME COMMAND
ps l	UID PID PPID CPU PRI NI VSZ RSS WCHAN STAT TT TIME COMMAND
ps u	USER PID %CPU %MEM VSZ RSS TT STAT STARTED TIME COMMAND
ps v	PID STAT TIME SL RE PAGEIN VSZ RSS LIM TSIZ %CPU %MEM COMMAND

有用的特殊组合	
ps	显示自己的进程
ps ax	显示所有进程
ps aux	显示所有进程，完整输出

图 26-5 ps 程序：BSD 选项

ps 程序显示系统中正在运行的进程的信息。**ps** 程序有两组选项可供使用：UNIX 选项和 BSD 选项。本表列举的是一些最重要的 BSD 选项。

假设您需要查看所有从您自己的终端上以您的用户标识正在运行的进程的基本信息。在这种情况下，您只需要输入这个命令本身：

ps

下面是使用 UNIX 版本的 **ps** 时的一些典型输出：

```

PID TTY      TIME CMD
2262 tty1    00:00:00 bash
11728 tty1    00:00:00 ps

```

下面是使用 BSD 版本的 **ps** 时的相同输出：

```

PID  TT  STAT   TIME  COMMAND
50384 p1  Ss    0:00.02  -sh (sh)
72883 p1  R+    0:00.00   ps

```

通常，**ps** 显示一个表，在这个表中，每行包含一个进程的信息。在上述 UNIX 示例中，我们看到的是进程#2262 和#11728 的信息。在上述 BSD 示范中，我们看到的是进程#50384 和#72883 的信息。

列表中的每列都包含一种具体类型的信息。根据使用哪些选项，列表会包含一些不同

类型的列。出于参考目的，图 26-6 列举了一些最常见的列标题。下面我们使用该图中的信息解释上述示范中的信息。

UNIX 标题	含义
ADDR	进程表中的虚拟地址
C	处理器利用率(废弃率)
CMD	正被执行的命令的名称
F	与进程相关的标志
NI	nice 值，用于设置优先级
PID	进程 ID
PPID	父进程的进程 ID
PRI	优先级(较大的数字=较低的优先级)
RSS	内存驻留空间大小(内存管理)
S	状态代码(D、R、S、T、Z)
STIME	累积系统时间
SZ	物理页的大小(内存管理)
TIME	累积 CPU 时间
TTY	控制终端的完整名称
UID	用户标识
WCHAN	等待通道

BSD 标题	含义
%CPU	CPU(处理器)使用百分比
%MEM	真实内存使用百分比
CMD	正被执行的命令的名称
COMMAND	正被执行的命令的完整名称
CPU	短期 CPU 使用(调度)
JOBC	作业控制统计
LIM	内存使用限额
NI	nice 值，用于设置优先级
PAGEIN	总的缺页错误(内存管理)
PGID	进程组号
PID	进程 ID
PPID	父进程的进程 ID
PRI	调度优先级
RE	内存驻留时间(单位为秒)
RSS	内存驻留空间大小(内存管理)

图 26-6 ps 程序：列标题

SESS	会话指针
SL	睡眠时间(单位为秒)
STARTED	定时启动
STAT	状态代码(O、R、S、T、Z)
TIME	累积 CPU 时间
TSIZ	文本大小(单位为 KB)
TT	控制终端的缩写名称
TTY	控制终端的完整名称
UID	用户标识
USER	用户名
VSZ	虚拟大小(单位为 KB)
WCHAN	等待通道

图 26-6 (续)

ps 程序显示进程的信息。所显示的信息按列进行组织，每列都有自己的标题。因为各个标题都是缩写，所以看上去可能有点神秘。

出于参考目的，本表列举了使用图 26-4 和图 26-5 中描述的基本选项时极有可能遇上的列标题。

大多数时候，可以忽略比较深奥的列。然而，为了满足您的好奇心，我对它们都进行了解释。可以看出，UNIX 选项使用的标题与 BSD 选项使用的标题有所不同。有关状态代码的含义，请参见图 26-7。

Linux、FreeBSD	
D	不可中断睡眠：等待事件结束(通常是 I/O，D=“磁盘”)
I	空闲：超过 20 秒的睡眠(仅适用于 FreeBSD)
R	正在运行或可运行(可运行=正在运行队列中等待)
S	可中断睡眠：等待事件结束
T	挂起：由作业控制信号挂起或者因为追踪而被挂起
Z	僵进程：终止后，父进程没有等待

Solaris	
O	正在运行：当前正在执行(O=“onproc”)
R	可运行：正在运行队列中等待
S	正在睡眠：等待事件结束(通常是 I/O)
T	挂起：由作业控制信号挂起或者因为追踪而被挂起
Z	僵进程：终止后，父进程没有等待

图 26-7 ps 程序：进程状态代码

对于特定的选项，**ps** 命令显示一系列数据，指示每个进程的状态。对于 UNIX 选项来说，该列标记为 **S**，而且包含一个单字符的代码。对于 BSD 选项来说，该列标记为 **STAT**，并且包含一个类似的代码，有时候后面还跟 1~3 个其他较不重要的字符。这里示范的是各个代码的含义，根据系统的不同，这些代码的含义可能稍微有所不同。

我们从前述的 UNIX 例子开始，该例显示的数据列有 4 个，分别为 **PID**、**TTY**、**TIME** 和 **CMD**。查阅图 26-6，可以得到如下信息。

PID: 进程 ID
TTY: 控制终端的名称
TIME: 累积 CPU 时间
CMD: 被执行命令的名称

因此，我们可以知道进程#2262 由终端 **tty1** 控制、几乎没有使用 CPU 时间，而且运行的是 **Bash**。进程#11728 的信息与此大致相同。唯一的区别就是这个进程正在运行 **ps** 命令。这个例子中看到的内容是最少的内容，因为至少会有两个进程在运行：自己的 **shell** 以及 **ps** 程序本身。但是，**ps** 进程的存活时间不长。实际上，一旦显示完输出，**ps** 程序就会死亡。

现在我们以相同的方式分析 BSD 的例子。BSD 例子中的输出有 5 个数据列，分别为 **PID**、**TT**、**STAT**、**TIME** 和 **COMMAND**。查阅图 26-6，可以得到如下信息。

PID: 进程 ID
TT: 控制终端的名称
STAT: 状态代码(O、R、S、T、Z)
TIME: 累积 CPU 时间
COMMAND: 被执行的完整命令

通常，BSD 版本的 **ps** 程序的输出相当直接，但是 **STAT** 列除外，我们稍后再详细解释这个列。

在结束本节之前，我希望展示一个小的但是有趣的变化：当使用 BSD 选项时，**ps** 显示缩写的终端名称。仔细看看上例中 **TT** 列的内容。注意该列的内容只有两个字符，在这个例子中是 **p1**。该终端的完整名称实际上是 **ttyp1**(终端名在第 23 章中讨论过)。

提示

比较 **ls** 和 **ps** 之间的相似性比较有趣。这两个程序都检查特定的数据结构，查看并显示相关信息。

ls 程序(参见第 25 章)检查 i 节点表(通过 i 节点号索引)，显示有关文件的信息。**ps** 程序检查进程表(通过进程 ID 索引)，显示有关进程的信息。

26.18 ps 程序：选择选项

使用 **ps** 的最佳方法就是先问两个问题：我对哪些进程感兴趣？我希望查看每个进程的哪些信息？一旦决定了您需要的内容，就可以借助图 26-4(UNIX 选项)或图 26-5(BSD 选项)选择合适的选项。

例如，假设您希望查看系统中正在运行的每个进程的进程 ID，以及所有父进程的进程

ID。我们首先从 UNIX 选项开始。首先，我们要问自己，哪个选项显示系统上所有的进程？从图 26-4 中可以知道这个选项是 **-e**(everything, 每个)。

接下来，必须查找显示每个进程及其父进程的进程 ID 的选项。通过查看图 26-6，我们知道希望得到的列标题是 **PID** 和 **PPID**。返回到图 26-4 中，查找显示这两个标题的选项。4 个选项都能完成这个作业，因此我们使用 **-f**(full output, 完整输出)，因为它显示的输出量最少。

将这些选项结合在一起，就得到了显示系统中正在运行的每个进程及其父进程的进程 ID 的方法：

```
ps -ef
```

很有可能，这条命令生成的输出会有许多行，所以最好将输出管道传送给 **less**(参见第 21 章)，从而每次一屏地显示输出：

```
ps -ef | less
```

下面对 BSD 版本的 **ps** 进程进行相同的分析。首先，查看图 26-5，检查哪个选项显示系统中所有的进程。这个选项就是 **ax**。接下来，查看显示父进程的进程 ID 的选项。我们有两个选择：**j** 和 **l**。最好选择 **j**，因为它生成的输出较少。因此，我们希望的 BSD 版本的命令为：

```
ps ajx | less
```

作为练习，下面我们尝试尽可能远地追踪进程的祖先。首先，使用 UNIX 版本的 **ps** 显示当前的进程：

```
ps
```

输出为：

```
PID TTY      TIME CMD
12175 tty2 00:00:00 bash
12218 tty2 00:00:00 ps
```

我们的目标是追踪 shell(进程#12175)的祖先。首先，我们问自己如下问题：哪个选项显示一个具体进程的信息？通过查看图 26-4，我们知道可以使用 **-p** 选项，后面跟进程 ID 来显示具体进程的信息。接下来再问自己，哪个选项显示父进程的进程 ID？答案是 **-f**。因此，我们使用如下命令进行搜索：

```
ps -f -p 12175
```

输出为：

```
UID      PID  PPID  C  STIME TTY      TIME CMD
harley  12175 1879  0 14:14 tty1     00:00:00 -bash
```

从这个输出中可以看出，进程#12175 的父进程是进程#1879。下面对新进程 ID 重复相

同的命令:

```
ps -f -p 1879
```

输出为:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1879	1	0	09:36	?	00:00:00	login -- harley

注意进程#1879 的父进程是进程#1。这是我们在本章前面讨论过的初始化进程。

在继续之前,我希望强调两点有趣的内容。首先,注意 **TTY** 列中的? 字符。这个字符表示该进程没有控制终端。我们称这样的进程为“守护进程”,本章后面将详细讨论守护进程。其次,我们知道进程#1879 正在以用户标识 **root** 运行程序 **login**。这是因为 **login** 是允许用户登录系统的程序。您是否还记得,在第 4 章中,我们曾使用该程序注销系统,以为新用户登录做好准备?

为了完成最终父进程的搜索,下面显示进程#1 的信息:

```
ps -f -p 1
```

输出为:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	09:34	?	00:00:01	init [5]

我们已经到达了家谱搜索的末尾。正如本章前面讨论的,进程#1(初始化进程)的父进程是进程#0(空闲进程)。顺便说一下,请注意在这里进程#1 运行 **init** 命令将系统启动到运行级别 5(多用户的 GUI 模式)。运行级别在第 6 章中讨论过。

26.19 ps 程序: 状态

下面通过讨论进程的状态展开对 **ps** 命令的讨论。正如本章前面讨论的,进程通常处于 3 种状态中的一个:在前台运行,在后台运行,或者挂起并等待恢复执行的信号。另外还有几种不太常见的变种,如僵状态,即进程已经死亡,但是进程的父进程没有等待该进程。

为了查看进程的状态,可以使用 **ps** 显示 **S** 列(对于 UNIX 选项来说)或者 **STAT** 列(对于 BSD 选项来说)。我们首先从 UNIX 选项开始。通过查看图 26-4,我们知道显示 **S** 列的 UNIX 选项是 **-l** 和 **-ly**。我们将使用 **-ly**,因为该选项显示的输出较少。因此,为了显示所有正在运行的进程以及它们的状态,可以使用:

```
ps -ly
```

下面是 Linux 系统的一些典型输出:

S	UID	PID	PPID	C	PRI	NI	RSS	SZ	WCHAN	TTY	TIME	CMD
S	500	8175	1879	0	75	0	464	112	wait	tty1	00:00:00	bash
T	500	8885	8175	0	75	0	996	366	finish	tty1	00:00:00	vim


```
R 500 9067 8175 2 78 0 996 077 -      tty1 00:00:02 find
R 500 9069 8175 0 78 0 800 034 -      tty1 00:00:00 ps
```

进程的状态由 **S** 列中的单字母代码描述。表示进程状态的代码的含义参见图 26-7。在这个例子中，可以看出列表中的第一个进程，即进程#8175(shell)的状态是代码 **S**。这意味着该进程正在等待某些事情完成(具体而言，该进程正在等待其子进程#90682，即 **ps** 程序本身)。

第二个进程#8885 的状态代码是 **T**，该代码意味着这个进程已被挂起。在这个例子中，当按下 **^Z** 键挂起 **vim** 编辑器时，**vim** 编辑器正运行在前台(这一过程在本章前面描述过)。

第三个进程#9067 的状态代码是 **R**，这意味着该进程正在运行。实际上，该进程就是正在后台运行的 **find** 命令(参见第 5 章)。

最后一个进程是 **ps** 程序本身。它的状态也是 **R**，因为它也正在运行——在这个例子中，在前台运行。该进程就是那个显示您正在阅读的输出的进程。实际上，当您看到输出时，这个进程已经终止，shell 进程(#8175)重新获得控制权。

在结束这个例子之前，再指出一些有趣的事情。通过查看 **PID** 和 **PPID**，我们可以知道 **shell** 是所有其他进程的父进程(您应该能理解这一点)。

下面讨论如何使用 **BSD** 选项查看进程的状态。首先，查看图 26-6，发现希望显示的标题是 **STAT**。接下来查看图 26-5，注意 **ps** 的所有缩写都显示 **STAT** 列，包括不含选项的 **ps** 命令自身。如果使用的是纯 **BSD** 系统，那么只需使用：

```
ps
```

如果使用的是混合系统(Linux 系统就是这样)，那么必须使用一个 **BSD** 选项强制进行 **BSD** 输出。我的建议是选择使用 **j** 选项，因为该选项生成的输出量最少：

```
ps j
```

下面是一些典型的输出，所使用的是 **FreeBSD** 系统上的纯 **ps** 命令(对于 **j** 选项来说，除了列多一些之外，输出与此相似)。

```
PID TT STAT    TIME COMMAND
52496 p0 Ss   0:00.02 -sh (sh)
52563 p0 T    0:00.02 vi test
54123 p0 Z    0:00.00 (sh)
52717 p0 D    0:00.12 find / -name harley -print
52725 p0 R+   0:00.00 ps
```

第一个进程#52496 是 **shell**。^{*}注意 **STAT** 列不止一个字符。其中第一个字符是状态代码。第二个字符说明的是一些深奥的技术信息，我们可以放心地忽略(如果您对这些信息感兴趣，则可以查看说明书页)。在这个例子中，状态代码是 **S**。通过查看图 26-7，我们知道该进程正在等待某些事情完成。具体而言，该进程正在等待其子进程#52725，即 **ps** 程序。

^{*} 第 11 章中讲过，旧 Bourne shell 的名称是 **sh**。您可能会感到奇怪，这是 Bourne shell 吗？答案是否定的，Bourne shell 已经多年没有使用了。这里是因为 **FreeBSD** shell 使用的名称碰巧是 **sh**。

第二个进程#52563 的状态代码是 **T**，这意味着该进程被挂起。在这个例子中，用户通过按[^]**Z** 键挂起了 **vi**。

第三个进程#54123，即一种老 shell，其状态代码是 **Z**，意味着该进程是僵进程。这种现象并不常见。僵进程就是不知何故，当进程死亡时，它的父进程没有等待(参见本章前面有关僵进程的讨论)。

第四个进程#52717 是一个在后台运行的 **find** 程序。它的状态代码是 **D**，表示它正在等待一个 I/O 事件结束(在这个例子中，即从磁盘读取数据)。这是合理的，因为 **find** 进程进行大量的 I/O 操作。但是，您必须记住，每当使用 **ps** 时，您查看的是一个瞬间的快照。在使用 **ps** 时，我们捕获到 **find** 正在等待 I/O。我们也可以容易地捕获到 **find** 的运行瞬间，在这种情况下，它的状态代码将是 **R**。

最后一个进程是#52725，即 **ps** 程序本身。它的状态代码是 **R**，因为它正在前台运行。

在结束这个例子之前，我再指出一件有趣的事情。如果查看输出的最右一列，即 **COMMAND** 列，会看到该列显示的是正被执行的命令的完整名称。该列仅在 BSD 选项中可用。对于 UNIX 选项来说，看到的只是 **CMD** 列，该列只显示名称，而不显示完整的命令。^{*}

提示

如果使用类似于 Linux 的同时支持 UNIX 选项和 BSD 选项的系统，那么您可以选择最适合自己的选项。例如，假设您希望以完整的命令名(**COMMAND**)，而不只是命令名(**CMD**)显示一串进程。如果您可以使用 BSD 选项，则可以使用命令：

```
ps j
```

当只使用 UNIX 选项时，就没有这么简单的方法。

偏执狂提示

在多用户系统上，可以通过使用 **ps** 偷看其他人正在做什么来自娱自乐。特别地，当使用 BSD 选项时，可以查看 **COMMAND** 列，即查看其他用户输入的完整命令名(如果您的系统不支持 BSD 选项，则可以使用 **w** 程序完成相同的事情，参见第 8 章)。

最初这看上去像是无伤大雅的娱乐，但是当您意识到系统上其他每个人都可能偷看您正在做什么时，情况就不是这样了。

因此，一定要小心。如果您是一个男孩，那么当系统管理员或者您的女朋友^{**}偷看您正在做什么时，看到您最近一小时输入的命令是 **vi pornography-list** 时，他们会怎么想呢？

26.20 监视系统进程：top、prstat

为了查看自己的进程，可以使用 **ps** 命令。但是，如果希望查看系统的整体运行情况，

^{*} 对于 Solaris 来说，**CMD** 列不显示完整的命令。

^{**} 如果您的女朋友是系统管理员，那么您需要更加小心。

该怎么办呢？可以确定的是，**ps** 有一些选项可以显示系统中所有进程的大量信息。但是，**ps** 有一个主要的限制：它只显示进程的静态快照，即瞬间的进程状态。因为进程是动态的，当需要查看各种进程如何不断地变化时，该限制就变得十分重要。在这种情况下，可以使用 **top** 程序每隔几秒钟显示整个系统的统计更新，并且实时显示最重要的进程的信息。

该程序的名称来源于一个事实，即它显示的是“顶端”进程，也就是那些使用最多 CPU 时间的进程。**top** 程序使用的语法稍微有点复杂，而且各个系统之间也有所不同。下面是 Linux 中 **top** 程序的基本语法。在其他系统中，选项会有所不同，因此在使用过程中一定要查看联机手册：

```
top [-d delay] [-n count] [-p pid[,pid]...]
```

其中 *delay* 是刷新闻隔(单位为秒)，*count* 是刷新的总时间量，*pid* 是进程 ID。

top 程序在大多数 Linux 和 BSD 系统中可用。如果您的系统没有 **top** 程序，那么该系统通常会提供等价的程序。例如，对于 Solaris 来说，可以使用 **prstat** 程序取代 **top** 程序。因为 **top** 程序的选项会根据其版本的不同而有所不同，所以最好花些时间查看系统上的说明书页。

为了查看 **top** 的工作机制，输入 **top** 命令本身：

top

在退出该程序时，可以按 **q** 键或 **^C** 键。

与 **less** 和 **vi** 相似，**top** 也以原始模式(参见第 21 章)工作。这允许它完全接管命令行和屏幕，根据需要显示行并改变字符。作为示例，请看图 26-8，该图是一些典型输出的缩减版本。

```
top - 9:10:24 up 14:50, 7 users, load average: 0.32,0.17,0.05
Tasks: 97 total, 1 running, 92 sleeping, 4 stopped, 0 zombie
Cpu(s): 1.7% us, 2.0% sy, 0.0% ni, 96.4% id, 0.0% wa
Mem: 385632k total, 287164k used, 98468k free, 41268k buffer
Swap: 786424k total, 0k used, 786424k free, 156016k cached

  PID USER  PR  NI  VIRT  RES  SHR  S  %CPU  %MEM  TIME+  COMMAND
 4016 harley 16   0 2124   992  780  R   1.3   0.3  0:00.35 top
3274 harley 15   0 7980 1808 1324  S   0.3   0.5  0:01.14 sshd
   1 root   16   0 1996   680  588  S   0.0   0.2  0:01.67 init
   2 root   34  19   0     0     0  S   0.0   0.0  0:00.00 ksoftirqd
   3 root   RT   0   0     0     0  S   0.0   0.0  0:00.00 watchdog
   4 root   10  -5   0     0     0  S   0.0   0.0  0:00.00 events
   5 root   10  -5   0     0     0  S   0.0   0.0  0:00.01 khelper
   6 root   11  -5   0     0     0  S   0.0   0.0  0:00.00 kthread
   8 root   10  -5   0     0     0  S   0.0   0.0  0:00.02 kblockd
  11 root   10  -5   0     0     0  S   0.0   0.0  0:00.00 khubd
```

图 26-8 top 程序

top 程序用来显示系统上“顶端”进程的动态信息，也就是使用最多 CPU 时间的进程。这里示范的是 **top** 程序的输出的缩减版本。该输出以一个固定间隔定期更新。在 **top** 执行时，可以键入命令控制它的行为。如果想获取帮助，可以按 **h** 键；如果想退出该程序，可以按 **q** 键或 **^C** 键。

top 程序的输出分成两部分。最上面的 5 行显示系统的总体信息。在我们的例子中，最顶端的那一行显示时间(9:10 AM)、系统已经运行了多长时间(14 小时 50 分钟)以及用户数量(7)。另外还有大量的其他更技术性的信息，显示有关进程、CPU 时间、真实内存(内存)以及虚拟内存(交换空间)的统计信息。

在系统信息下面，是描述各个进程的数据，按照 CPU 的使用顺序，每行描述一个进程。在我们的例子中，系统并不忙碌。实际上，**top** 本身是最顶端的进程。

top 程序功能强大，因为它定期自动地刷新统计信息。默认的时间间隔根据 **top** 程序版本的不同而不同。例如，在我的 Linux 系统上，默认时间间隔是 3 秒；在我的 FreeBSD 系统上，默认时间间隔是 2 秒；在我的 Solaris 系统(使用 **prstat**)上，默认时间间隔是 5 秒。如果要改变刷新频率，则可以使用 **-d(delay, 延迟)** 选项。例如，告诉 **top** 程序每隔 1 秒刷新一次，可以使用：

```
top -d 1
```

一些版本的 **top** 甚至允许输入更短的时间间隔。如果系统支持这种版本的 **top**，则可以尝试运行一个非常快的刷新频率，例如：

```
top -d 0.1
```

在繁忙的系统上，这会产生一种迷人的显示效果。^{*}

因为 **top** 以原始模式工作，所以可以在程序运行时键入不同的命令。最重要的命令是 **q**，该命令退出程序。下一个最重要的命令是 **h(help, 帮助)** 或 **?**，该命令显示所有命令的汇总列表。第三条命令是 **<Space>** 键，但该命令通常不会记录。该命令强制 **top** 立即刷新显示。当选择的是较慢的刷新频率，而又需要立即更新显示时，按 **<Space>** 键就非常有用。这里并不会详细介绍所有的命令，因为它们的技术性都相当强。但是，当您有时间时，可以按 **h** 键，查看您自己的 **top** 提供有哪些命令。

如果要进行额外的控制，则还有另外两个选项可以使用。默认情况下，**top** 一直不停地进行刷新。**-n** 选项告诉 **top** 只在特定的时间内进行刷新。例如，为了刷新显示 6 次，每次间隔 10 秒，可以使用：

```
top -d 10 -n 6
```

在这个例子中，该程序将只运行 60 秒。

为了显示一个具体进程的信息，可以使用 **-p** 选项，后面跟进程 ID，例如：

```
top -p 3274
```

当需要指定多个进程 ID 时，可以用逗号将它们分开。例如，下述命令使用一个 1 秒的刷新频率，显示 1 号进程至 5 号进程：

```
top -d 1 -p 1,2,3,4,5
```

通常，系统管理员和程序员比常规用户对 **top** 的使用要多。一般情况下，管理员使用 **top** 进行性能监测。例如，管理员可能希望查看新应用程序在服务器上干什么，或者希望评估两个不同的数据库程序，查看哪个效率高。程序员通常使用 **top** 测试程序在不同的工

^{*} 对男孩的提示：如果您有一个重要的约会，并且热切希望给对方留下深刻的印象，那么您可以邀请她到您的住所，让她坐在您的计算机前面。然后登录一个繁忙的 Unix 或 Linux 系统，并以 1 秒或更低的刷新频率运行 **top**。如果这不能给她留下深刻的印象，那就没有什么希望了。

作负载下的执行情况。

您将会发现 **ps** 程序要比 **top** 程序更适合您的需求。但是，在特定的场合，**top** 程序价值无限。例如，如果您正在使用的系统突然之间变得异常缓慢，那么您可以使用 **top** 程序查看系统到底发生了什么事情。

26.21 显示进程树：pstree、ptree

到目前为止，我们已经讨论了两个可以显示进程信息的重要工具：查看静态信息的 **ps** 和查看动态信息的 **top**。查看进程信息的第三个工具是 **pstree**，该工具对于理解进程之间的关系非常有用。

在本章前面，我解释过每个进程(除了最初的第一个进程)都由另一个进程创建。当发生这种事情时，原始进程称为父进程，新创建的进程称为子进程。无论何时，当创建新进程时，新进程都会获得一个称为进程 ID 或 PID 的标识号。

在启动过程的末尾，内核创建那个最初的进程，即空闲进程，该进程的 PID 是 0。在执行了一系列任务之后，空闲进程创建第二个进程，即初始化进程，该进程的 PID 为 1。然后空闲进程永久地休眠(因此它这样命名)。

初始化进程的任务就是创建一系列其他进程。大多数的第三代进程都是守护进程(我将在本章后面解释这个名称)，守护进程的任务就是等待某些事情的发生，然后进行适当的响应。具体而言，有一些守护进程不做其他事情，专门等待用户的登录。当用户准备登录时，守护进程就创建另一个进程来处理这一任务。然后登录守护进程创建另一个进程运行用户的 shell。最后，每当 shell 需要执行用户的程序时，shell 就会创建另一个进程来执行任务。

尽管这种安排看上去很复杂，但是它可以通过一个简单的事实极大地简化，即每个进程(除了第一个)都只有一个父进程。因此，可以这样假设，即系统中的所有进程都被安排到一个大型的树型层次结构中，而初始化进程位于树的根部。我们称这样的数据结构为进程树(process tree)，并使用它们来表示父进程及其子孙之间的连接。

您可以使用 **pstree** 程序显示系统进程树的任何部分。例如，从初始化进程开始显示整个进程树，或者基于特定的 PID 或用户标识显示一个子树。这里使用的语法为：

```
pstree [-aAcGnpu] [ pid | userid ]
```

其中 *pid* 是进程 ID，*userid* 是用户标识。

大多数 Unix 系统都提供有 **pstree** 程序。如果您的系统中没有提供 **pstree** 程序，那么这些系统中有时候会提供等价的程序。例如，在 Solaris 中，就可以使用 **ptree**(详情请参见联机手册)。在其他系统上，**ps** 命令拥有显示进程树的特殊选项。您可以试一试 **ps f** 或 **ps -H**，但是这些命令的输出可能没有 **pstree** 程序那么出色。

为了查看 **pstree** 的工作机制，我们首先输入不带任何选项的命令。默认情况下，**pstree** 从初始化进程开始，绘制整个系统的进程树。这会生成许多行输出，因此最好将输出管道传送给 **less**(参见第 21 章)，从而每次显示一屏。

```
pstree | less
```

下面是一个缩减版的例子，示范一个 Linux 系统的前 8 行输出。注意该树的根——初始化进程——位于该图的顶端。

```
init-+-apmd
      |-and
      |-automount
      |-crond
      |-cups-config-daemon
      |-cupsd
      |-2*[dbus-daemon---(dbus-daemon)]
      |-dbus-launch
```

观察这个进程树时，我希望您注意几点。第一，在每一层，树都根据进程名称按字母顺序排列。这是默认设置，您可以使用 **-n** 选项改变(参见下面)。

接下来，注意倒数第二行中的表示法 **2***。这意味着有两个完全相同的子树。使用这样的表示方法允许 **pstree** 创建一张更紧凑的图表。如果希望 **pstree** 扩展所有的子树，包括完全相同的子树，则可以使用 **-c**(do not compact, 不要压缩)选项。

最后要注意的是，**pstree** 使用纯 ASCII 字符绘制树的分支。对于一些终端来说，**pstree** 将使用特殊行绘制字符。如果基于某些原因，输出看上去不正确，则可以使用 **-A** 选项强制 **pstree** 使用 ASCII 字符，或者使用 **-G** 选项强制 **pstree** 使用行绘制字符。花一些时间体验一下下述命令，查看哪种类型的输出最适合自己的系统：

```
pstree -A | less
pstree -G | less
```

除了显示选项以外，**pstree** 还有其他选项可以用来控制显示的信息。我最喜欢的两个选项就是 **-p**(显示每个进程的 PID)和 **-n**(按 PID，而不是按进程名称对进程排序)：

```
pstree -np
```

下面是使用这两个选项的输出的前 8 行。可以看出，该进程树从进程#1(初始化进程)开始。该进程有许多子进程：进程#2、#3、#4、#5、#6 等。进程#6 还拥有自己的子进程：#8、#11、#13、#80 等。

```
init(1)-+-ksoftirqd(2)
          |-watchdog(3)
          |-events(4)
          |-khelper(5)
          |-kthread(6)-+-kblockd(8)
                        |
                        |   |-khubd(11)
                        |   |-kseriod(13)
                        |   |-pdflush(80)
```

默认情况下，**pstree** 从根进程(也就是说，从进程#1)开始绘制整个进程树。但是，有

时候我们只对整个进程树的一部分感兴趣。这种情况下，我们可以采取两种方法来限制输出。如果指定 PID，那么 **pstree** 将显示由该特定进程派生的子树。

下面举例说明。您的 shell 是 Bash。从这个 shell 开始，您在后台有两个进程 **make** 和 **gcc**。此外，您还有两个挂起的进程 **vim** 和 **man**。您希望显示一个只展现这几个进程的进程树。首先，您使用 **ps** 或 **echo \$\$** 查看自己的 shell 的 PID。该 shell 的 PID 是 #2146。然后，您输入下述命令：

```
pstree -p 2146
```

下面是输出：

```
bash--+-gcc(4252)
      |-pstree(4281)
      |-make(4276)
      |-man(4285)---sh(4295)---less(4301)
      `--vim(4249)
```

注意 **man** 已经创建了一个子进程来运行一个新 shell(#4295)，而这个 shell 又创建了另一个子进程(#4301)来运行 **less**。这是因为 **man** 调用了 **less** 来显示它的输出。

限制进程树范围的第二种方法就是指定一个用户标识而不是 PID。当这样做时，**pstree** 只显示以该用户标识运行的进程，例如：

```
pstree -p harley
```

我希望提及的最后两个选项通常用来与进程名称一起显示额外的信息。**-a**(all, 全部)选项显示每个进程的整个命令行，而不只是程序的名称。**-u**(userid change, 用户标识改变)选项标识当子进程以不同于父进程的用户标识运行时所发生的改变。

26.22 思考 Unix 如何组织进程和文件：fuser

在继续讨论之前，我希望暂停一下，提醒您想一想 Unix 组织进程和文件之间的相似性。

进程和文件都可以被认为是层次结构树中已有的，在层次结构树中，树位于顶端。进程树的根是进程 #1(初始化进程)。文件树的根是根目录(参见第 23 章)。在进程树中，每个进程只有一个父进程，而且父进程位于子进程的上面。在文件树中，每个子目录只有一个父目录，而且父目录位于子目录的上面。为了显示文件树，我们可以使用 **tree** 程序(参见第 24 章)。为了显示进程树，我们可以使用 **pstree** 程序。

再深入一点，我们还可以发现更多的相似性。每个进程都由一个唯一的称为进程 ID 的数字标识。从内部讲，Unix 通过使用由进程 ID 索引的进程表记录进程。在进程表中，每个条目都包含一个进程的信息。同样，Unix 通过使用由 i 节点号索引的 i 节点表记录文件。在 i 节点表中，每个条目(i 节点)包含一个文件的信息。

但是，基于一个很好的理由，我们不能将两者相提并论。为什么呢？因为进程和文件之间存在一个基本的区别。进程是动态的：每个时刻都在变化，描述进程的数据一直在变

化而文件相对比较静止。

例如, 为了显示文件的信息, 我们使用的是 **ls** 程序(参见第 24 章和第 25 章), 该程序只是简单地在 i 节点表中查看数据。为了显示进程的信息, 我们使用的是 **ps** 程序和 **top** 程序, 而且进程信息的收集比较困难。可以确定的是, 在进程表中可以查看一些基本数据。但是, 大多数动态信息必须从内核中获取, 而这些信息的获取就不像查看表那样简单。

为了获得完成作业所需的数据, **ps** 和 **top** 程序必须使用一种称为 **proc** 文件(参见第 23 章)的伪文件。在 **/proc** 目录中, 每个进程都用自己的 **proc** 文件表示。当程序需要进程的信息时, 程序就读取该进程的 **proc** 文件, 并最终触发一个向内核请求提供所需数据的请求。整个事情的发生非常快, 从而使进程信息的查看看上去并不比文件信息的查看更复杂。

您可能想知道, 有没有工具可以在进程和文件之间搭起一座桥梁? 这样的工具确实存在。其中一个最有趣的工具就是 **fuser**, 这是一个系统管理工具, 用来列举所有正在使用一个给定文件的进程。例如, 假设您输入了下述命令运行 **find** 程序(第 25 章), 查找名为 **foo** 的文件。注意该程序在后台运行, 而且将标准输出重定向到文件 **bar***中:

```
find / -name foo -print > bar 2>/dev/null &
```

当该程序启动时, shell 显示下述消息, 包括作业 ID(3)和进程 ID(3739):

```
[3] 3739
```

因为标准输出被重定向到 **bar**, 所以当程序运行时, 这个特定的文件就被使用。为了检查这一点, 可以输入下述命令:

```
fuser bar
```

下面是输出:

```
bar: 3739
```

可以看出, 文件 **bar** 被进程#3739 使用。通过这种方式, **fuser** 提供了一个有关一个单独的工具如何同时收集进程和文件信息的有趣例子。

如果您想自己尝试一下 **fuser**, 那么您可能会遇到一个值得讨论的问题。**fuser** 程序按道理由系统管理员使用。基于这个原因, 该程序通常和其他此类工具一样存储在一个管理目录中, 如 **/sbin**(参见第 23 章)。但是, 除非以超级用户登录, 否则管理目录极有可能没有位于您的搜索目录中。这意味着, 当您键入 **fuser** 命令时, shell 查找不到这个程序。

当遇到这样的问题时, 只需使用 **whereis**(参见第 25 章)查找系统中 **fuser** 程序的位置。例如:

```
whereis fuser
```

下面是一些典型的输出:

```
fuser: /sbin/fuser /usr/share/man/man1/fuser.1.gz
```

在这个例子中, 第一个路径是程序的位置, 第二个路径是说明书页的位置。为了运行

* 有关名称 **foo** 和 **bar** 的讨论, 请参见第 9 章。

fuser, 所需做的全部就是向 **shell** 说明在何处查找这个程序:

```
/sbin/fuser bar
```

当希望运行不在自己搜索路径中的程序时, 可以使用这种技巧。

26.23 杀死进程: **kill**

kill 程序有两种应用: 终止进程以及向进程发送信号。在本节中, 我们将讨论进程的终止。在下一节中, 我们再讨论有关发送信号的内容。

通常, 程序一直运行, 直至程序自己结束运行或者告诉它们退出。我们也可以通过按 **^C** 键发送 **intr** 信号(参见第7章), 或者通过键入退出命令提早停止程序。但是, 这些方法并不总是起作用。例如, 有时候, 程序是冻结的, 停止响应键盘。在这种情况下, 通过按 **^C** 或者键入退出命令就不起作用。当希望终止正在后台运行的程序时也会出现类似的问题。因为后台进程不从键盘读取, 所以键盘信号无法直接到达这种程序。

在这些情形中, 可以使用 **kill** 程序终止程序。当通过这种方式终止程序时, 我们称杀死程序。所使用的语法为:

```
kill [-9] pid... | jobid...
```

其中 *pid* 或 *jobid* 用来标识进程。

大多数时候, 我们希望的是杀死一个单独的进程。这时, 我们首先使用 **ps** 或 **jobs** 查看希望杀死进程的进程 ID 或作业 ID, 然后使用 **kill** 执行进程的实际终止操作。考虑下面的例子。假设您已经输出下述命令在后台运行 **make** 程序:

```
make game > makeoutput 2> makeerrors &
```

过了一会儿, 您决定杀死该进程。第一步就是查找该进程的进程 ID, 所以输入:

```
ps
```

输出为:

PID	TTY	TIME	CMD
2146	tty2	00:00:00	bash
5505	tty2	00:00:00	make
5534	tty2	00:00:00	ps

我们希望的进程 ID 是 **5505**。为了杀死该进程, 输入:

```
kill 5505
```

shell 将杀死该进程, 并显示一个消息, 例如:

```
[2] Terminated make game >makeoutput 2>makeerrors
```

这意味着运行 **make game** 的程序已经被杀死。消息行开头的数字意味着该进程为作业#2。另一种列举进程的方法是使用 **jobs -l** 命令。假设我们使用下述命令来取代 **ps**:

```
jobs -l
```

下面是可能看到的输出:

```
[2]- 5505 Running make game >makeoutput 2>makeerrors. &
```

我们同样可以使用 **kill 5505** 来杀死 **make** 进程。但是, 还有另外一种方法。这种方法以使用 **fg** 和 **bg** 命令(参见图 26-3)时的相同方式指定作业号。因此, 在这个例子中, 下述任何命令都可以完成任务:

```
kill 5505
kill %-
kill %2
kill %make
kill %?game
```

下面是另一种常见的情形。一个前台进程失去了响应, 无论键入任何内容, 包括 **^C** 都无法停止这个程序。您有两个选择。第一, 可以尝试着按 **^Z** 键挂起该进程。如果这种方法能够成功, 那么您可以使用 **ps** 或 **jobs** 查找该进程, 然后使用 **kill** 终止该进程。

另外一种方法就是打开一个新的终端窗口, 并使用 **ps -u** 或 **ps U** 列举以您的用户标识运行的所有进程。然后确定失控的进程, 并使用 **kill** 终止该进程。实际上, 当进程自身失去控制时, 这有时候是杀死进程的唯一一种可行的方法。

如果使用的是远程 Unix 主机, 则可能有第三种选择。如果所有其他方法都已经失败, 则只需简单地断开与主机的连接。在一些系统上, 当您断开您与主机的连接时, 内核会自动杀死您所有的进程。当然, 这也将杀死其他正在运行的程序。

无论何时, 当杀死有子进程的进程时, 也会将子进程杀死。因此, 通过查看并杀死原始的父进程, 就可以简单地杀死一组相关的进程(在 Unix 中, 家族深深地绑定在一起)。

当 **^C** 键或者退出命令不起作用时, **kill** 通常可以完成任务。但有时候, 即便是 **kill** 也会不起作用。在这种情况下, 通常采用另外一种变体: 指定选项 **-9** 作为命令的一部分。这将发送“确定杀死”信号 9(我们将在下一节中讨论)。例如:

```
kill -9 5505
kill -9 %2
```

发送信号 9 永远可以起作用。但是, 这应该是最后一种选择, 因为它杀死进程的过程非常快。使用 **kill -9** 杀死进程时, 并不允许进程释放正在使用的资源。例如, 进程可能不关闭文件(可能导致数据丢失)、释放内存等。使用 **kill -9** 还可能导致不受约束的子进程, 而这种进程不能正确死亡(参见本章前面有关孤儿的讨论)。

尽管内核通常会清理这种混乱, 但是在诉诸于激烈的措施之前最好还是尝试应用其他的技术。

26.24 向进程发送信号：kill

正如前面所讨论的，可以使用 **kill** 程序终止其他方法到达不了的进程。但是，**kill** 程序不只是一个终止工具。它实际上还是一个功能强大的程序，可以向任何进程发送任何信号。当以这种方式使用 **kill** 时，更通用的语法为：

```
kill [-signal] pid...|jobid...
```

其中 *signal* 是希望发送的信号类型，*pid* 或 *jobid* 用来标识进程，和前一节中讨论的相同。

在第 23 章中，我们讨论了进程间通信(interprocess communication, IPC)的概念，即两个进程之间数据的交换。那时，我们讨论了采用命名管道作为从一个进程向另一个进程发送数据的方法。**kill** 程序的目的是支持一种不同类型的 IPC，具体而言，就是发送一个非常简单的消息，这种消息称为信号。信号只不过是发送给进程的一个数字，从而让进程知道发生了哪些类型的事情。进程会识别信号，从而决定执行什么动作。当进程这样做时，我们称进程捕获了信号。

在第 7 章中，在几种特殊键组合(如 **^C** 和 **^Z**)的讨论过程中，我们介绍了信号。当按下这样一个这样的键组合时，就会向当前的前台进程发送一个信号。例如，当按下 **^C** 键时，就发送信号 2。

Unix 中使用了大量的信号，其中大多数只有系统程序员才会感兴趣。出于参考目的，图 26-9 列举了一些最常用的信号。注意，每个信号都由一个数字标识，另外还有一个标准化的名称和缩写(都采用大写字母键入)。

编号	名称	缩写	描述
1	SIGHUP	HUP	中止：注销或者终端失去连接时发送给进程
2	SIGINT	INT	中断：当按下 ^C 键时发送
9	SIGKILL	KILL	杀死：立即终止，进程不能捕获
15	SIGTERM	TERM	终止：请求终止，进程不能捕获
18	SIGCONT	CONT	继续：恢复挂起的进程，由 fg 或 bg 发送
19	SIGSTOP	STOP	停止(挂起)：当按下 ^Z 键时发送

图 26-9 信号

信号用作简单的，但是重要的进程间控制形式。本列表列举了一些最常用的信号，以及这些信号的名称。当使用 **kill** 向进程发送信号时，可以使用信号的编号、名称或者缩写指定。如果使用名称或缩写，则一定要键入大写字母。各个系统之间的信号编号可能有所不同，所以通常最好使用名称或缩写，这些都是标准化的。这里示范的信号编号是 Linux 系统中使用的。

通常，**HUP**、**INT**、**KILL** 和 **TERM** 的信号编号在所有系统上都是相同的。但是，其他信号的编号可能在不同类型的 Unix 系统上有所不同。基于这一原因，最好是使用信号的名称或缩写——它们总是相同的，而不是使用编号。图 26-9 中示范的信号编号是 Linux 系统使用的。

如果希望查看自己系统所支持的全部信号列表，则可以输入 **kill** 命令，后面跟 **-l(list)**，

列举)选项:

```
kill -l
```

如果自己的系统不支持这个选项,则可以查找包含文件(参见第 23 章)signal.h,并显示这个文件的内容。命令可以采用下述形式之一:

```
locate signal.h
find / -name 'signal.h' -print 2> /dev/null
```

kill 程序允许指定任何自己希望的信号。例如,假设您希望挂起作业%2,该作业在后台运行。为此,只需向这个作业发送信号 STOP:

```
kill -STOP %2
```

如果不指定信号,那么 kill 将默认发送 TERM 信号。因此,下述命令(都作用于进程 3662)是等价的:

```
kill 3662
kill -15 3662
kill -TERM 3662
kill -SIGTERM 3662
```

正如前面所述,信号有许多,而 kill 命令的目的就是向一个特定的进程发送一个具体的信号。从这个意义上讲,可能最好将这个命令命名为 signal。但是,默认情况下,kill 发送的是 TERM 信号,而该信号的作用就是杀死进程,这就是将该命令命名为 kill 的原因。实际上,大多数人在使用 kill 时只是用来杀死进程,而不是用来发送信号。

出于安全考虑,常规的用户标识只能向自己的进程发送信号。但是,超级用户允许向系统上的任何进程发送信号。这意味着,如果您使用的是自己的系统,当遇到一个不死的进程时,您可以切换到超级用户,使用 kill 杀死这个进程。但是一定小心,因为“超级用户+kill”是一个高度致命的组合,如果您不知道自己正在做什么,则有可能使您陷入极大的麻烦。

26.25 设置进程的优先级: nice

在本章的开头已解释过即使小型的 Unix 系统也有可能同时拥有超过一百个进程在运行。大型的系统可能有数千个进程,这些进程都需要共享系统的资源:处理器、内存、I/O 设备、网络连接等。为了管理这样复杂的工作负载,内核使用了一个称为调度器的复杂子系统,调度器的任务就是在各个进程之间动态地分配资源。

在生成瞬间的决策过程中,调度器认为每个进程都关联有一系列不同的值。其中一个较重要的值就是优先级,优先级用来指示一个进程相对于其他进程有多大的优先权。优先级根据一系列因素设置,这些通常会超出个人用户的控制范围。原因有两方面。

首先,有效地管理进程是一个非常复杂的操作,而调度器能够比人——即便是有经验的系统管理员——更快更好地完成这项任务。其次,如果能够管理优先级,则用户可能会

试图在牺牲其他用户和系统本身利益的情况下提升自己程序的优先级。

但是,在特定的场合中,也有可能希望做相反的事情。也就是说,可能希望降低某个程序的优先级。一般情况下,当需要在较长时间内运行且要求大量 CPU 时间的程序时,就会这样做。在这种情况下,您可能是一位高尚的人,以一个较低的优先级在后台运行该程序。毕竟,您并不在意该程序花更长的时间来运行,以一个较低的优先级运行该程序允许调度器将高优先级赋予其他程序,从而使系统的响应更快,更加高效。

为了以一个较低的优先级运行程序,可以使用一个称为 **nice** 的工具(您知道这个名称的来源吗?)该工具的语法为:

```
nice [-n adjustment] command
```

其中 *adjustment* 是一个数值, *command* 是希望运行的命令。

使用 **nice** 的最简单方式就是在准备后台运行的命令前面简单地键入该命令的名称,例如:

```
nice gcc myprogram.c &
```

当以这种方式启动程序时, **nice** 将使程序以一个较低的优先级运行。但是, **nice** 不会自动地在后台运行程序。如果希望在后台运行程序,则需要在命令的后面键入一个 **&** 字符。

哪些类型的程序能够和 **nice** 一起使用呢?通常,任何可以在后台运行并且使用大量 CPU 时间的程序都可以和 **nice** 一起使用。**nice** 的传统应用是那些编译大量源代码、生成软件包或者执行复杂数学计算的程序。例如,如果您共享一台多用户系统,而且在测试一个计算 1 000 000 位 π 的程序,您肯定希望这个程序以一个尽可能低的优先级运行。

在使用 **nice** 过程中,有两个警告需要引起注意。第一, **nice** 只能应用于自己独立存在的程序。例如,可以对外部命令和 shell 脚本应用 **nice**。但是,内置 shell 命令(内部命令)、管道线或复合命令不能降低优先级。

第二个考虑就是只能对后台运行的程序使用 **nice**。尽管也可以降低前台程序的优先级,但是这样做没有意义。毕竟,当程序在前台运行时,您希望该程序能够尽可能快地响应。

在大多数情况下,按照前面描述的方法使用 **nice** 就已经足够了。但有时候,可能希望对优先级的降低方式有更多的控制。为此,可以使用 **-n** 选项,后面跟一个 **nice** 值。在大多数系统上,可以指定从 0~19 的 **nice** 值。**nice** 值越高,程序的优先级就越低(这意味着您就是一位越高尚的用户)。

当以普通方式(不使用 **nice**)运行程序时,赋予程序的 **nice** 值为 0。这被认为是普通优先级。当不带 **-n** 选项使用 **nice** 时, **nice** 值默认为 10,正好位于优先级范围的中间。大多数时候,这可以满足全部需求。但是,如果需要的话,也可以指定自己的 **nice** 值。

例如,假设您有一个程序 **calculate**,该程序需要花费数小时的时间来执行复杂的数学计算。作为一个高尚的人,您决定在后台以一个尽可能低的优先级运行这个程序。您可以使用一个类似于下述命令的命令:

```
nice -n 19 calculate > outputfile 2> errorfile &
```

这时候您可能想知道,如果一个高的 **nice** 值将降低程序的优先级,那么一个低的 **nice** 值会提升程序的优先级吗?答案是肯定的,但仅当您是超级用户时情况才是如此。作为超级用户,您可以指定一个 -20 到 -1 之间的负值。例如,为了以尽可能高的优先级运行一

个非常特殊的程序，可以切换到超级用户，并输入命令：

```
nice -n -20 specialprogram
```

可以想象，实际上极少需要设置一个负的 `nice` 值。实际上，大多数时候，最好是让调度器管理系统中程序的优先级。

提示

当您共享一台多用户系统时，最好是养成使用 `nice` 在后台以一个低优先级运行 CPU 密集型程序的习惯。这可以防止这类程序降低系统对其他用户的响应速度。

但是，`nice` 也可以在单用户系统上提供便利。当强制后台程序以低优先级运行时，就可以防止它们降低紧要工作的速度。

(换句话说，做高尚的人总是有回报的。)

26.26 改变现有进程的优先级：renice

有时候，您可能发现自己等待一个在前台运行的程序花费的时间太长，这时您觉得将这个程序以一个低的优先级在后台运行更合理。在这种情形中，您需要做的就是按下[^]Z 键挂起这个进程，使用 `bg` 将该进程移到后台，然后降低它的优先级。为了降低现有进程的优先级，可以使用 `renice` 命令。它的语法为：

```
renice niceness -p processid
```

其中 *niceness* 是 `nice` 值，*processid* 是进程 ID。

正如上一节中讨论的，较高的 `nice` 值意味着较低的优先级。当以常规用户使用 `renice` 时，只允许提高进程的 `nice` 值，而不能降低。也就是说，您只可以降低进程的优先级，而不能提升进程的优先级。另外，作为一个合理的防护措施，常规用户只能改变自己进程的 `nice` 值(您能明白原因吗？)但是，这些限制并不适用于超级用户。

下面举一个如何使用 `renice` 的例子。输入下述命令，运行一个计算 1 000 000 位 π (π) 的程序：

```
picalculate > outputfile 2> errorfile
```

在观察这个程序一段时间，发现它没有动静后，您决定让这个程序在后台运行。同时，最好还尽可能降低这个程序的优先级。首先，按[^]Z 键挂起前台进程，您将看到一个类似于下面的消息：

```
[1]+ Stopped picalculate >outputfile 2>errorfile
```

这告诉您进程(即作业#1)已经挂起。现在您可以使用 `bg` 将该进程移至后台：

```
bg %1
```

然后您会看到如下消息，该消息告诉您该程序已经在后台运行：

```
[1]+ picalculate >outputfile 2>errorfile &
```

接下来, 使用 **ps** 查看该进程的进程 ID:

```
ps
```

输出为:

```
PID TTY      TIME CMD
4052 tty1    00:00:00 bash
4089 tty1    00:00:00 picalculate
4105 tty1    00:00:00 ps
```

最后, 使用 **renice** 给予该进程尽可能高的 nice 值, 从而尽可能降低该进程的优先级:

```
renice 19 -p 4089
```

您将看到类似于下面的消息:

```
4089: old priority 0, new priority 19
```

提示

如果您的系统没有任何停顿的理由, 那么您可以使用 **top** 查看是否有非交互式进程占用了大量的 CPU 时间。如果是这样, 那么可以考虑使用 **renice** 降低这些进程的优先级(警告: 不要乱动自己不理解的进程)。

26.27 守护进程

您可能想知道自己的系统目前有多少个进程正在运行, 这很容易实现。只需使用 **ps** 程序, 加上合适的选项列举每个进程, 每行一个进程, 然后将输出管道传送给 **wc -l**(参见第 18 章)统计行数即可。

下面是两条完成这一作业的命令。第一条命令对 **ps** 使用 UNIX 选项。第二条命令使用 BSD 选项:

```
ps -e | wc -l
ps ax | wc -l
```

作为测试, 我在 3 种不同的 Unix 系统上运行这两条命令。首先, 我查看的是 Solaris 系统, 我通过 Internet 访问该系统。该系统绝对没有其他人在使用, 也没有运行其他内容。接下来, 我查看自己身边的 Linux 系统, 该系统正在运行一个完全基于 GUI 的桌面环境, 除了我自己外, 没有其他人使用这个系统。最后, 我查看的是 FreeBSD 系统, 该系统充当一个中等规模的 Web 服务器和数据库服务器。

下面是我的发现。不考虑 **ps** 程序本身, 小型的 Solaris 系统运行 46 个进程, 小型的 Linux 系统运行 95 个进程, 而中等的 FreeBSD 系统运行 133 个进程。可以这样说, 这些都还是相对较小的数字。大型的 Unix 系统同时运行数百个甚至数千个进程并不罕见。

很明显, 大多数进程并不是由用户运行的程序。那么这些进程是什么呢? 答案就是它们是守护进程, 即在后台运行的程序, 完全不与任何终端连接, 目的是提供服务(对于 Microsoft Windows, 相同类型的功能由所谓的“服务”程序提供)。一般情况下, 守护进程

静静地后台等待某些事情的发生：事件、请求、中断、给定的时间间隔等。当触发发生时，守护进程进入动作，完成执行作业所需的事情。

守护进程执行运行系统所需的大量任务。出于参考目的，我在图 26-10 中列举了一些比较有趣的守护进程。尽管在许多 Unix 系统上，您只会看到少数几个守护进程(如 **init**)，但是守护进程有许多变体。为了查看系统上的守护进程，可以使用 **ps** 命令，寻找输出的 **TTY** 列中的 **?** 字符——这表示该进程不受终端的控制。

守护进程	目的
init	其他所有进程的祖先，收养孤儿
apache	Apache Web 服务器
atd	运行 at 程序排列的作业
crond	管理预调度作业的执行(cron 服务)
cupsd	打印调度器(CUPS=Common Unix Printing System, 通用 Unix 打印系统)
dhcpd	为客户端动态配置 TCP/IP 信息(DHCP)
ftpd	FTP 服务器(FTP=File Transfer Protocol, 文件传输协议)
gated	网络的网关路由
httpd	Web 服务器
inetd	Internet 服务
kerneld	根据需要加载或卸载内核模块
kudzu	在启动过程中检测并配置新/改变过的硬件
lpd	打印队列(行打印机守护进程)
mysql	MySQL 数据库服务器
named	Internet DNS 域名服务器(DNS=Domain Name System, 域名系统)
nfsd	网络文件访问(NFS=Network File System, 网络文件系统)
ntpd	时间同步(NTP=Network Time Protocol, 网络时间协议)
rpcbind	远程过程调用(RPC)
routed	管理网络路由表
sched	swapper 的另一个名称
sendmail	SMTP 服务器(电子邮件)
smbd	Windows 客户端的文件共享&打印服务(Samba)
sshd	SSH(安全 shell)连接
swapper	将数据从内存复制到交换空间，回收物理内存
syncd	文件系统与系统内存内容的同步
syslogd	收集各种系统消息(系统日志记录器)
xinetd	Internet 服务(取代 inetd)

图 26-10 守护进程

守护进程是一种在后台静静运行，不与任何终端相连，提供服务的进程。Unix 系统通常有许多守护进程，每个守护进程都等待在需要时执行任务。这里列举的是众多系统上一些最有趣的守护进程。注意守护进程的许多名称都以字母“d”结尾。

最好的命令就是使用 `ps` 的变体，显示所有不受终端控制的进程：

```
ps -t - | less
```

如果该命令不适用于自己的系统，则可以试一试下述命令：

```
ps -e | grep '?' | less
```

大多数守护进程是在启动过程中的最后一部分自动创建的。在一些情形中，这些进程由初始化进程(进程#1)创建。在其他情形中，这些进程由终止自身的父进程创建，从而使守护进程成为孤儿。本章前面讲过，所有的孤儿都被初始化进程收养。因此，不管是哪一种方式，许多守护进程最终都是进程#1 的孩子。基于这一原因，守护进程的一个定义就是任何没有控制终端的后台进程，它们的父进程的进程 ID 为 #1。

如果使用的是 Linux 系统，则要花一些时间来看看 `/etc/rc.d/init.d` 目录。在这个目录中有大量的 shell 脚本，每个 shell 脚本都用来启动、停止或者重新启动一个特定的守护进程。

名称含义

守护进程

守护进程是那些通过在后台静静运行来提供服务，并且不与任何终端相连的进程。尽管该名称的发音是“dee-mon”，但是其正确的拼法是“daemon”。

您可能偶尔读到该名称代表“Disk and Executing Monitor(磁盘及执行监视器)”，一个来自古老的 DEC 10 及 20 计算机的术语。但是，这种解释是虚构的，并不符合事实。名称“daemon”由麻省理工学院的程序员首次使用，那时他们在 1963 年开发的 CTSS(Compatible Time-sharing System, 兼容分时系统)上工作。他们创造了这个术语，指在 ITS(Incompatible Time-sharing System, 非兼容分时系统)上工作的其他程序员所指的“dragon”。

CTSS 和 ITS 都是 Unix 的祖先。ITS 尽管陌生，但它是一个重要的操作系统，在麻省理工学院曾经风靡一时。因为 CTSS 和 ITS 程序员是从麻省理工学院转到贝尔实验室的，所以守护进程的思想也随之迁了过去。

那么为什么命名为“守护进程”呢？故事之一就是这一名称起源于“Maxell's demon”，这是一个由苏格兰物理学家 James Maxell(1831-1879)设计的与第二热力学定律相关的虚构人物。您可以相信它，也可以不相信它(我不相信)。

无论起源于何处，没有人知道我们为什么要使用英国的变体拼法。在凯尔特神话中，daemon 通常是良性词或中性词，只是一种精神或灵感。但是，demon 总是一个恶魔实体。或许这里面有深层的含义。

26.28 结束语

感谢您挤出如此众多的时间与我一起讨论 Unix 和 Linux。我编写本书的目的是使那些聪明人理解 Unix，非常荣幸有机会为大家提供力所能及的帮助。

Unix 已经吸引了许多天才的计算机用户和程序员，这些人非常乐意使用 Unix。Unix 如此神奇的原因之一就是在那些西装革履的人们开始注意并使用 Unix 之前，Unix 的大部分就已经设计完成。这就是 Unix 工作如此出色，如此优美的原因。基本的 Unix 设计原则在商业和市场人员开始试图从 Unix 赚钱之前很久就已经开发完成。正如第 2 章中讨论的，在 20 世纪 90 年代，这一基本原则被移植到 Linux 项目中和开放源代码社区，并取得了神奇的结果。

前面讲过 Unix 并不容易学习，但是容易使用。到现在为止，您应该意识到这句话的含义：Unix 是一个为聪明人设计的重要工具，而不是那些每天卸载拱石的人首次使用时也能方便掌握的简单工具。

您要记住，每次学习 Unix 和使用 Unix 时都将有极大的回报。在您工作时，我可能无法在您的身边，但是您可以拥有本书，而且我已经尽了最大的努力在使用 Unix 方面为您提供最大的帮助。

尽管我会不时地调侃一下——实际上，每当能够开玩笑时我都会这样做，但是编辑们可能无法理解——我希望大家都是最优秀的。在您阅读或再次阅读本书时，请记住：我就站在您的身边。

提示

Unix 极其有趣。

26.19 练习

1. 复习题

1. 什么是进程？操作系统的哪一部分管理进程？定义下述术语：进程 ID、父进程、子进程、分叉和执行。

2. 什么是作业？操作系统的哪一部分管理作业？什么是作业控制？

在前台运行作业与在后台运行作业有什么区别？如何在前台运行作业？如何在后台运行作业？如何将作业从前台移到后台？

3. **ps** 程序用来显示进程的信息。该程序可以使用哪两种类型的选项？对于每种类型的选项，显示下述信息需要使用哪条命令？

- 您当前的进程
- 进程#120357
- 系统中运行的所有进程
- 与用户标识 **weedly** 关联的进程

4. 您是一名系统管理员。有一个系统看上去好像停顿了，您的任务就是了解原因。首先，您希望查看系统上正在运行的各个进程，以及各个进程如何不停地变化。您应该使用哪个程序呢？指定运行这个程序的命令，该命令每隔 5 秒自动刷新一次。

5. 杀死进程和停止进程之间有什么区别？如何杀死进程？如何停止进程？
6. 假设您已经启动了一个程序 **foobar**，该程序的运行已经失控。杀死这个程序需要采取什么步骤呢？如果 **foobar** 没有响应，又该怎么办呢？

2. 练习题

1. 输入一条暂停 5 秒钟的命令行，然后显示消息 “I am smart.” 等待 5 秒确信该命令已经运行。

现在将延迟改变为 30 秒，并重新输入该命令行。这一次，在用尽 30 秒之前，按下 ^C 键。这又会发生什么情况呢？为什么？

2. 您刚通过 Internet 使用用户标识 **weedly** 登录到一个 Unix 系统。输入命令 **ps -f**，看到如下结果：

UID	PID	PPID	C	STIME	TTY	TIME	CMD
weedly	2282	2281	0	15:31	pts/3	00:00:00	-bash
weedly	2547	2282	0	16:13	pts/3	00:00:00	ps -f

事情一切正常。但是出于好奇心，您决定查看系统的其他部分，因此输入命令 **ps -af**。在众多输出行中，有如下一行：

```
weedly 2522 2436 0 16:09 pts/4 00:00:00 vim secret
```

有人以您的用户标识登录了系统！您该怎么办呢？

3. 创建一个管道线，统计系统中守护进程的数量。然后创建第二个管道线，显示所有守护进程的一个有序列表。输出时，只需显示守护进程的名称，不必显示其他内容。

3. 思考题

1. 通过使用 **kill** 命令杀死进程要比想象的复杂。描述一种较简单的方法来实现同样的功能。
2. **ps** 为什么有两种不同类型的选项？这是好事还是坏事呢？

Unix 命令一览表

本附录描述了书中所涉及的 143 个 Unix 命令。在每个名称的后面，方括号中的数字表示讨论该命令的参考章号。

要按照命令的类别查询书中所涉及 Unix 命令的一览表，请参见附录 B。

! [13]	重新执行历史列表中的命令
!! [13]	重新执行历史列表中的最后一条命令
& [26]	在后台运行程序
^^ [13]	替换/重新执行历史列表中的最后一条命令
^Z [26]	挂起(暂停)前台进程
alias [13]	创建/显示别名
apropos [9]	基于键盘搜索显示命令名
bash [11]	Bash shell
bc [8]	任意精度，易于使用的计算器
bg [26]	将作业移至后台
bindkey [13]	设置命令行编辑模式
cal [8]	显示一个日历
calendar [8]	从 calendar 文件中显示当前提醒
cat [16]	组合文件；将标准输入复制到标准输出
cd [24]	改变工作目录
chmod [25]	改变文件或目录的文件权限
chsh [11]	改变默认 shell
cmp [17]	比较两个文件
colrm [16]	删除指定的数据列
comm [17]	比较两个有序文件，显示区别
cp [25]	复制文件；复制目录
csh [11]	C-Shell
cut [17]	提取指定的数据列/字段

date [8]	显示时间和日期
dc [8]	任意精度, 基于栈的计算器
df [24]	显示文件系统已使用/可使用的磁盘空间
diff [17]	比较两个文件, 显示区别
dirs [24]	显示/清除目录栈的内容
dmesg [6]	显示启动信息(Linux)
du [24]	显示文件使用的磁盘空间量
dumpe2fs [24]	显示超块的文件系统信息
echo [12]	将参数写到标准输出
env [12]	显示环境变量
exit [4]	退出 shell
expand [18]	将制表符更改为空格
export [12]	将 shell 变量输出到环境中
fc [13]	显示/重新执行历史列表中的命令
fg [26]	将作业移至前台
file [24]	分析文件的类型
find [25]	在目录树中搜索文件, 处理结果
fmt [18、22]	格式化段落, 从而使它们看上去更漂亮
fold [18]	将长行格式化为较短的行
fuser [26]	识别使用给定文件的进程
grep [19]	选择包含指定模式的行
groups [25]	显示用户标识所属的组
head [16、21]	从数据的开头选择行
hexdump [21]	显示二进制(非文本)文件
history [13]	显示历史列表中的命令
hostname [8]	显示系统的名称
id [25]	显示当前用户标识和组标识
info [9]	从 Info 参考系统中显示文件
init [6]	切换到另一个运行级别
jobs [26]	显示作业的信息
join [19]	基于共同字段组合数据列
kill [26]	终止进程; 向进程发送信号
ksh [11]	Korn shell

last [4]	查看用户标识上一次登录的时间
leave [8]	在指定的时间显示一个提醒
less [21]	分页程序：每次一屏地显示数据
ln [25]	创建文件的一个新链接
locate [25]	搜索文件
lock [8]	临时锁定终端
login [4]	终止登录 shell 并初始化一个新登录
logout [4]	终止登录 shell
look [19]	选择以指定模式开头的行
ls [24、25]	显示文件的各种类型的信息
man [9]	显示 Unix 联机参考手册的页面
mkdir [24]	创建目录
mkfifo [23]	创建命名管道
more [21]	分页程序：每次一屏地显示数据
mount [23]	挂载文件系统
mv [24、25]	移动或重命名文件或目录
nice [26]	使用指定的调度优先级运行程序
nl [18]	在文本中添加行号
od [21]	显示二进制(非文本)文件
passwd [4]	改变登录口令
paste [17]	组合数据列
popd [24]	改变工作目录：将名称从目录栈中弹出
pr [18]	将文本格式化成页面或列
print [12]	将参数写到标准输出
printenv [12]	显示环境变量
prstat [26]	显示进程的动态信息
ps [26]	显示进程的信息
pstree [26]	显示进程树图表
ptree [26]	显示进程树图表
pushd [24]	改变工作目录，将名称压入到目录栈中
pwd [24]	显示工作目录的路径名
quota [8、24]	显示系统资源限额
reboot [6]	重新启动计算机
renice [26]	改变已运行程序的调度优先级

rev [16]	数据每行中的字符反向排列
rm [25]	删除文件或目录
rmdir [24]	删除空目录
sdiff [17]	比较两个文件, 显示区别
sed [19]	非交互式文本编辑
set [12]	设置/显示 shell 选项和 shell 变量
setenv [12]	设置/显示环境变量
sh [11]	Bourne shell
shred [25]	安全删除文件
shutdown [6]	关闭计算机
sleep [26]	延迟一个指定的时间间隔
sort [19]	排序数据; 查看数据是否是有序的
split [16]	将大文件分隔成小文件
stat [25]	显示 i 节点的信息
strings [19]	在二进制文件中搜索字符串
stty [7]	设置/显示终端的操作选项
su [6]	改变到超级用户或另一个用户标识
sudo [6]	以超级用户运行一条单独的命令
suspend [26]	挂起(暂停)shell
tac [16]	组合文件的同时将文本行的顺序反转
tail [16、21]	在数据的末尾选择行
tcsh [11]	Tcsh shell
tee [15]	将标准输入复制到文件和标准输出
top [26]	显示使用最多 CPU 的进程的数据
touch [25]	更新文件的访问/修改时间; 创建文件
tr [19]	改变或删除选择的字符或字符串
tree [24]	显示目录树的图表
tsort [19]	由偏序创建一个全序
tty [23]	显示表示终端的特殊文件的名称
type [8]	定位命令: 显示命令的路径名或别名
umask [25]	在文件创建过程中设置文件模式掩码
umount [23]	卸载文件系统
unalias [13]	删除别名
uname [8]	显示操作系统的名称
unexpand [18]	将空格改变为制表符
uniq [19]	移除文本文件中相临的重复行

unset [12]
unsetenv [12]
uptime [8]
users [8]

vi [22]
view [22]
vim [22]

w [8]
wc [18]
whatis [9]
whence [8]
whereis [25]
which [8]
who [8]
whoami [8]

xargs [25]
xman [9]

删除 shell 变量
删除环境变量
显示系统已经运行的时间
显示当前登录到系统的用户标识

vi 文本编辑器
以只读模式启动 **vi** 文本编辑器
Vim 文本编辑器

显示用户标识和活动进程的信息
统计行数、单词数和字符数
为指定命令显示一行摘要信息
定位命令：显示命令的路径名或别名
查看与命令关联的文件
定位命令：显示命令的路径名或别名
显示当前登录的用户标识的信息
显示当前登录的用户标识

使用来自标准输入的参数运行命令
基于 GUI，显示联机参考手册的页面



Unix 命令分类表

本附录摘要描述了书中所涉及的 143 个 Unix 命令，并且按照命令的类别进行排列。在每个名称的后面，方括号中的数字表示讨论该命令的参考章号。按照字母顺序排列的命令列表参见附录 A。

Unix 命令分类如下：

构建块
命令工具
比较文件
目录
显示数据
文档资料
编辑

文件
文件系统
登录和注销
进程和作业控制
Shell
选择数据

系统工具
终端
文本格式化
工具
用户和用户标识
变量

构建块

cat [16]
tee [15]
xargs[25]

组合文件，将标准输入复制到标准输出
将标准输入复制到文件和标准输出
使用来自标准输入的参数运行命令

命令工具

alias [13]
type [8]
unalias [13]
whence [8]
which [8]

创建/显示别名
定位命令：显示命令的路径名或别名
删除别名
定位命令：显示命令的路径名或别名
定位命令：显示命令的路径名或别名

比较文件

cmp [17]

比较两个文件

comm [17]	比较两个有序文件, 显示区别
diff [17]	比较两个文件, 显示区别
sdiff [17]	比较两个文件, 显示区别

目录

cd [24]	改变工作目录
chmod [25]	改变文件或目录的文件权限
dirs [24]	显示/清除目录栈中的内容
du [24]	显示文件使用的磁盘空间量
file [24]	分析文件的类型
ls [24、25]	显示文件的各种类型的信息
mkdir [24]	创建目录
mv [24、25]	移动或重命名文件或目录
popd [24]	改变工作目录, 将名称从目录栈中弹出
pushd [24]	改变工作目录, 将名称压入到目录栈中
pwd [24]	显示工作目录的路径名
rm [25]	删除文件或目录
rmdir [24]	删除空目录
tree [24]	显示目录树的图表

显示数据

cat [16]	组合文件, 将标准输入复制到标准输出
echo [12]	将参数写到标准输出
head [16、21]	从数据的开头选择行
hexdump [21]	显示二进制(非文本)文件
less [21]	分页程序: 每次一屏地显示数据
more [21]	分页程序: 每次一屏地显示数据
od [21]	显示二进制(非文本)文件
print [12]	将参数写到标准输出
tail [16、21]	在数据的末尾选择行

文档资料

apropos [9]	基于键盘搜索显示命令名
info [9]	从 Info 参考系统中显示文件
man [9]	显示 Unix 联机参考手册的页面
whatis [9]	为指定命令显示一行摘要信息

xman [9] 基于 GUI，显示联机参考手册的页面

编辑

sed [19] 非交互式文本编辑
vi [22] **vi** 文本编辑器
view [22] 以只读模式启动 **vi** 文本编辑器
vim [22] Vim 文本编辑器

文件

chmod [25] 改变文件或目录的文件权限
cp [25] 复制文件；复制目录
du [24] 显示文件使用的磁盘空间量
find [25] 在目录树中搜索文件，处理结果
ln [25] 创建文件的一个新链接
locate [25] 搜索文件
ls [24、25] 显示文件的各种类型的信息
mkfifo [23] 创建命名管道
mv [24、25] 移动或重命名文件或目录
rm [25] 删除文件或目录
shred [25] 安全删除文件
stat [25] 显示 i 节点的信息
touch [25] 更新文件的访问/修改时间；创建文件
umask [25] 在文件创建过程中设置文件模式掩码
whence [8] 定位命令：显示命令的路径名或别名
whereis [25] 查看与命令关联的文件

文件系统

df [24] 显示文件系统已使用/可使用的磁盘空间
dumpe2fs [24] 显示超块的文件系统信息
mount [23] 挂载文件系统
umount [23] 卸载文件系统

登录和注销

login [4] 终止登录 shell 并初始化一个新登录
logout [4] 终止登录 shell

passwd [4]

改变登录口令

进程和作业控制

& [26]

在后台运行程序

^Z [26]

挂起(暂停)前台进程

fg [26]

将作业移至前台

suspend [26]

挂起(暂停)shell

jobs [26]

显示作业的信息

bg [26]

将作业移至后台

ps [26]

显示进程的信息

top [26]

显示使用最多 CPU 的进程的数据

prstat [26]

显示进程的动态信息

pstree [26]

显示进程树图表

ptree [26]

显示进程树图表

fuser [26]

识别使用指定文件的进程

kill [26]

终止进程; 向进程发送信号

nice [26]

使用指定的调度优先级运行程序

renice [26]

改变已运行程序的调度优先级

Shell

! [13]

重新执行历史列表中的命令

!! [13]

重新执行历史列表中的最后一条命令

^^ [13]

替换/重新执行历史列表中的最后一条命令

bash [11]

Bash shell

bindkey [13]

设置命令行编辑模式

chsh [11]

改变默认 shell

csh [11]

C-Shell

exit [4]

退出 shell

fc [13]

显示/重新执行历史列表中的命令

history [13]

显示历史列表中的命令

ksh [11]

Korn shell

sh [11]

Bourne shell

tcsh [11]

Tcsh shell

选择数据

cut [17]

提取指定的数据列/字段

grep [19]	选择包含指定模式的行
head [16、21]	从数据的开头选择行
look [19]	选择以指定模式开头的行
strings [19]	在二进制文件中搜索字符串
tail [16、21]	在数据的末尾选择行

系统工具

dmesg [6]	显示启动信息(Linux)
hostname [8]	显示系统的名称
init [6]	切换到另一个运行级别
reboot [6]	重新启动计算机
shutdown [6]	关闭计算机
su [6]	改变到超级用户或另一个用户标识
sudo [6]	以超级用户运行一条单独的命令
uname [8]	显示操作系统的名称
uptime [8]	显示系统已经运行的时间

终端

lock [8]	临时锁定终端
stty [7]	设置/显示终端的操作选项
tty [23]	显示表示终端的特殊文件的名称

文本格式化

colrm [16]	删除指定的数据列
expand [18]	将制表符更改为空格
fmt [18、22]	格式化段落，从而使它们看上去更漂亮
fold [18]	将长行格式化为较短的行
join [19]	基于共同字段组合数据列
nl [18]	在文本中添加行号
paste [17]	组合数据列
pr [18]	将文本格式化成页面或列
rev [16]	数据每行中的字符反向排列
sed [19]	非交互式文本编辑
split [16]	将大文件分隔成小文件
tac [16]	组合文件的同时将文本行的顺序反转
tr [19]	改变或删除选择的字符或字符串

unexpand [18]	将空格改变为制表符
uniq [19]	移除文本文件中相邻的重复行

工具

bc [8]	任意精度，易于使用的计算器
cal [8]	显示一个日历
calendar [8]	从 calendar 文件中显示当前提醒
date [8]	显示时间和日期
dc [8]	任意精度，基于栈的计算器
leave [8]	在指定的时间显示一个提醒
sleep [26]	延迟一个指定的时间间隔
sort [19]	排序数据；查看数据是否是有序的
tsort [19]	由偏序创建一个全序
wc [18]	统计行数、单词数和字符数

用户和用户标识

groups [25]	显示用户标识所属的组
id [25]	显示当前用户标识和组标识
last [4]	查看用户标识上一次登录的时间
quota [8、24]	显示系统资源限额
users [8]	显示当前登录到系统的用户标识
w [8]	显示用户标识和活动进程的信息
who [8]	显示当前登录的用户标识的信息
whoami [8]	显示当前登录的用户标识

变量

echo [12]	将参数写到标准输出
env [12]	显示环境变量
export [12]	将 shell 变量输出到环境中
print [12]	将参数写到标准输出
printenv [12]	显示环境变量
set [12]	设置/显示 shell 选项和 shell 变量
setenv [12]	设置/显示环境变量
unset [12]	删除 shell 变量
unsetenv [12]	删除环境变量

vi 命令小结

本附录包含的是本书中所涉及的全部 **vi** 命令。更多的信息请参见第 22 章，这一章详细讨论了 **vi**。

启动

vi file	启动 vi ，编辑指定的文件
vi -R file	以只读模式启动 vi ，编辑指定的文件
view file	以只读模式启动 vi ，编辑指定的文件
vim file	启动 Vim ，编辑指定的文件
vim -C file	以兼容模式启动 vi

停止

:q!	不保存数据而停止
ZZ	保存数据并停止
:wq	保存数据并停止
:x	保存数据并停止

系统故障后恢复

vi -r	显示可以恢复的文件的名称
vi -r file	启动 vi ，恢复指定的文件

修订键

<Backspace>/<Delete>	删除键入的最后一个字符
^W	删除键入的最后一个单词
^X/^U	删除整行

控制显示

^L	重新显示当前屏幕
:set number	显示内部行号
:set nonumber	不显示内部行号

移动光标

h	将光标向左移动一个位置
j	将光标向下移动一个位置
k	将光标向上移动一个位置
l	将光标向右移动一个位置
<Left>	将光标向左移动一个位置
<Down>	将光标向下移动一个位置
<Up>	将光标向上移动一个位置
<Right>	将光标向右移动一个位置
<Backspace>	将光标向左移动一个位置
<Space>	将光标向右移动一个位置
-	将光标移动到上一行的开头
+	将光标移动到下一行的开头
<Return>	将光标移动到下一行的开头
0	将光标移动到当前行的开头
\$	将光标移动到当前行的末尾
^	将光标移动到当前行的第一个非空格/制表符
w	将光标移动到下一个单词的词首
e	将光标移动到下一个单词的词尾
b	将光标移动到上一个单词的词首
W	同 w ; 忽略标点符号
E	同 e ; 忽略标点符号
B	同 b ; 忽略标点符号
)	将光标移动到下一个句子的开头
(将光标移动到上一个句子的开头
}	将光标移动到下一个段落的开头
{	将光标移动到上一个段落的开头
H	将光标移动到屏幕的第一行
M	将光标移动到屏幕的中间行
L	将光标移动到屏幕的最后一行

在编辑缓冲区中移动

^F	向下(前)移动一屏
^B	向上(后)移动一屏
n^F	向下移动 <i>n</i> 屏
n^B	向上移动 <i>n</i> 屏
^D	向下移动半屏
^U	向上移动半屏
n^D	向下移动 <i>n</i> 行
n^U	向上移动 <i>n</i> 行

搜索模式

/regex	向前搜索指定的正则表达式
/	向前重复搜索上一模式
?regex	向后搜索指定的正则表达式
?	向后重复搜索上一模式
n	重复上一个/或? 命令, 方向相同
N	重复上一个/或? 命令, 方向相反

正则表达式中使用的特殊字符

.	匹配除换行符之外的任何单个字符
*	匹配 0 个或多个前面的字符
^	匹配行的开头
\$	匹配行的末尾
\<	匹配单词的开头
\>	匹配单词的末尾
[]	匹配括号中包含的任何一个字符
[^]	匹配不在括号中的任何字符
\	按照字面意义解释接下来的符号

行号

nG	跳转到行号 <i>n</i>
1G	跳转到编辑缓冲区中的第一行
gg	跳转到编辑缓冲区中的第一行
G	跳转到编辑缓冲区中的最后一行

:map g 1G	定义宏, 使 g 和 1G 拥有相同效果
:n	跳转到行号 <i>n</i>
:1	跳转到编辑缓冲区中的第一行
:\$	跳转到编辑缓冲区中的最后一行

插入

i	切换到插入模式: 在光标位置前插入
a	切换到插入模式: 在光标位置后插入
I	切换到插入模式: 在当前行开头插入
A	切换到插入模式: 在当前行末尾插入
o	切换到插入模式: 打开当前行下面一行
O	切换到插入模式: 打开当前行上面一行
<Escape>	离开插入模式, 切换到命令模式

进行修改

r	只替换 1 个字符(不进入输入模式)
R	键入中进行替换
s	通过插入替换 1 个字符
C	通过插入从当前光标替换到行末
cc	通过插入替换整个当前行
S	通过插入替换整个当前行
cmove	通过插入从当前光标替换到 <i>move</i> 指定的位置
~	改变字母的大小写

替换模式

:s/pattern/replace/	替换当前行
:lines/pattern/replace/	替换指定行
:line,lines/pattern/replace/	替换指定范围
:%s/pattern/replace/	替换全部行

在命令的末尾, 使用 **c** 可以用来请求确认, **g(global, 全局)** 可以用来替换每行的全部匹配结果。指定行号时, 可以使用一个实际数字, 也可以使用 **.**(句点) 表示当前行, 或者 **\$**(美元符号) 表示编辑缓冲区中的最后一行。数字 **1** 表示编辑缓冲区中的第一行。

撤销或重复改变

u	撤销修改编辑缓冲区的上一条命令
----------	-----------------

U	恢复当前行
.	重复修改编辑缓冲区的上一条命令

分隔和连接行

r<Return>	将当前行分隔成两行(将字符替换成新行字符)
J	将当前行与下一行连接成一个长行
:set wm=<i>n</i>	在离右边缘 <i>n</i> 个位置时自动换行

删除

x	删除光标处字符
X	删除光标左边的字符
D	从当前光标删除到行尾
dd	删除整个当前行
dmove	从当前光标删除到 <i>move</i> 所指定的位置
dG	从当前行删除到编辑缓冲区的末尾
d1G	从当前行删除到编辑缓冲区的开头
lined	删除指定的行
:line,lined	删除指定的范围

删除：有用的组合

dw	删除 1 个单词
dnw	删除 <i>n</i> 个单词
dnW	删除 <i>n</i> 个单词(忽略标点符号)
db	向后删除 1 个单词
dn)	删除 <i>n</i> 个句子
dn}	删除 <i>n</i> 个段落
dG	从当前行删除到编辑缓冲区的末尾
dgg	从当前行删除到编辑缓冲区的开头
d1G	从当前行删除到编辑缓冲区的开头

复制上一次删除

P	复制缓冲区；插入到当前光标后面/下面
P	复制缓冲区；插入到当前光标前面/上面
xp	调换两个字符
deep	调换两个单词(从第一个单词的左边开始)

ddp	调换两行
"1pu.u.u...	恢复某一次删除

复制与移动行

:linecotarget	复制指定的行，在目标下面插入
:line,linecotarget	复制指定的范围，在目标下面插入
:linemtarget	移动指定的行，在目标下面插入
:line,linemtarget	移动指定的范围，在目标下面插入

接出

ymove	将当前光标接出到 <i>move</i>
yy	接出整个当前行

接出：有用的组合

yw	接出 1 个单词
ynw	接出 <i>n</i> 个单词
ynW	接出 <i>n</i> 个单词(忽略标点符号)
yb	向后接出 1 个单词
yn)	接出 <i>n</i> 个句子
yn}	接出 <i>n</i> 个段落
yG	从当前行接出到编辑缓冲区的末尾
ygg	从当前行接出到编辑缓冲区的开头
y1G	从当前行接出到编辑缓冲区的开头

执行 shell 命令

!:command	暂停 vi ，执行指定的 shell 命令
!! pause vi,	执行上一条 shell 命令
:sh	暂停 vi ，启动一个 shell
!:csh	暂停 vi ，启动一个新的 C-Shell

向编辑缓冲区读入数据

:liner file	在指定行后插入 <i>file</i> 的内容
:r file	在当前行后插入 <i>file</i> 的内容
:liner !command	在指定行后插入 <i>command</i> 的输出

:r !command	在当前行后插入 <i>command</i> 的输出
:r !look pattern	插入以指定模式开头的单词

使用 shell 命令处理数据

n !!command	对 <i>n</i> 行数据执行 <i>command</i>
!move command	对当前光标至 <i>move</i> 所指定的位置的数据执行 <i>command</i>
!move fmt	格式化当前光标到 <i>move</i> 所指定的行

写入数据

:w	将数据写入到原始文件
:w file	将数据写入到指定文件
:w>> file	将数据追加到指定文件

在编辑过程中改变文件

:e file	编辑指定的文件
:e! file	编辑指定的文件，忽略自动检查

缩写

:ab short long	设置 <i>short</i> 为 <i>long</i> 的缩写
:ab	显示当前缩写
:una short	取消缩写 <i>short</i>



ASCII 码

在 20 世纪 90 年代之前，Unix(以及大多数计算机系统)使用的字符编码是 ASCII 码，通常简称为 ASCII。该名称代表“American Standard Code for Information Interchange，美国信息交换标准码”。

ASCII 码创建于 1967 年。ASCII 为每个字符指定 7 位，因此总共可以表示 128 个字符。这些位组合的范围为 0000000(即十进制 0)至 1111111(即十进制 127)。基于这一原因，128 个 ASCII 字符的编号为 0 至 127。

构成 ASCII 码的 128 个字符包含 33 个“控制字符”和 95 个“可显示字符”。控制字符的讨论请参见第 7 章。可显示字符如下所示，包括 52 个字母(26 个大写字母、26 个小写字母)、10 个数字、32 个标点符号以及空白符(位于下述列表的第一个位置)：

```
!"#$%&'()*+,-./0123456789:;<=>?  
@ABCDEFGHIJKLMNPQRSTUVWXYZ[\]^_  
'abcdefghijklmnopqrstuvwxyz{|}~
```

为了方便起见，大多数 Unix 系统都有一个参考页，列出了全部的 ASCII 码，从而使您在需要时可以快速地查看 ASCII 码。然而，ASCII 码参考页并不是标准化的，因此显示 ASCII 码参考页的方式根据所使用系统的不同而有所不同。详情请参见图 19-1 或图 20-5。

字符	十进制	十六进制	八进制	二进制	
	0	00	000	0000 0000	(null)
Ctrl-A	1	01	001	0000 0001	
Ctrl-B	2	02	002	0000 0010	
Ctrl-C	3	03	003	0000 0011	
Ctrl-D	4	04	004	0000 0100	
Ctrl-E	5	05	005	0000 0101	
Ctrl-F	6	06	006	0000 0110	
Ctrl-G	7	07	007	0000 0111	(嘀嘀声)
Ctrl-H	8	08	010	0000 1000	backspace
Ctrl-I	9	09	011	0000 1001	tab

(续表)

字符	十进制	十六进制	八进制	二进制	
Ctrl-J	10	0A	012	0000 1010	
Ctrl-K	11	0B	013	0000 1011	
Ctrl-L	12	0C	014	0000 1100	
Ctrl-M	13	0D	015	0000 1101	return
Ctrl-N	14	0E	016	0000 1110	
Ctrl-O	15	0F	017	0000 1111	
Ctrl-P	16	10	020	0001 0000	
Ctrl-Q	17	11	021	0001 0001	
Ctrl-R	18	12	022	0001 0010	
Ctrl-S	19	13	023	0001 0011	
Ctrl-T	20	14	024	0001 0100	
Ctrl-U	21	15	025	0001 0101	
Ctrl-V	22	16	026	0001 0110	
Ctrl-W	23	17	027	0001 0111	
Ctrl-X	24	18	030	0001 1000	
Ctrl-Y	25	19	031	0001 1001	
Ctrl-Z	26	1A	032	0001 1010	
Ctrl-[27	1B	033	0001 1011	escape
Ctrl-\	28	1C	034	0001 1100	
Ctrl-]	29	1D	035	0001 1101	
Ctrl-^	30	1E	036	0001 1110	
Ctrl-`	31	1F	037	0001 1111	
(space)	32	20	040	0010 0000	space
!	33	21	041	0010 0001	(感叹号)
"	34	22	042	0010 0010	(双引号)
#	35	23	043	0010 0011	(井号)
\$	36	24	044	0010 0100	(美元符号)
%	37	25	045	0010 0101	(百分比符号)
&	38	26	046	0010 0110	(和号)
'	39	27	047	0010 0111	(单引号)
(40	28	050	0010 1000	(左圆括号)
)	41	29	051	0010 1001	(右圆括号)
*	42	2A	052	0010 1010	(星号)

(续表)

字符	十进制	十六进制	八进制	二进制	
+	43	2B	053	0010 1011	(加号)
,	44	2C	054	0010 1100	(逗号)
-	45	2D	055	0010 1101	(减号/连字符)
.	46	2E	056	0010 1110	(句点)
/	47	2F	057	0010 1111	(斜线)
0	48	30	060	0011 0000	
1	49	31	061	0011 0001	
2	50	32	062	0011 0010	
3	51	33	063	0011 0011	
4	52	34	064	0011 0100	
5	53	35	065	0011 0101	
6	54	36	066	0011 0110	
7	55	37	067	0011 0111	
8	56	38	070	0011 1000	
9	57	39	071	0011 1001	
:	58	3A	072	0011 1010	(冒号)
;	59	3B	073	0011 1011	(分号)
<	60	3C	074	0011 1100	(小于号)
=	61	3D	075	0011 1101	(等号)
>	62	3E	076	0011 1110	(大于号)
?	63	3F	077	0011 1111	(问号)
@	64	40	100	0100 0000	(at 符号)
A	65	41	101	0100 0001	
B	66	42	102	0100 0010	
C	67	43	103	0100 0011	
D	68	44	104	0100 0100	
E	69	45	105	0100 0101	
F	70	46	106	0100 0110	
G	71	47	107	0100 0111	
H	72	48	110	0100 1000	
I	73	49	111	0100 1001	
J	74	4A	112	0100 1010	
K	75	4B	113	0100 1011	

(续表)

字符	十进制	十六进制	八进制	二进制	
L	76	4C	114	0100 1100	
M	77	4D	115	0100 1101	
N	78	4E	116	0100 1110	
O	79	4F	117	0100 1111	
P	80	50	120	0101 0000	
Q	81	51	121	0101 0001	
R	82	52	122	0101 0010	
S	83	53	123	0101 0011	
T	84	54	124	0101 0100	
U	85	55	125	0101 0101	
V	86	56	126	0101 0110	
W	87	57	127	0101 0111	
X	88	58	130	0101 1000	
Y	89	59	131	0101 1001	
Z	90	5A	132	0101 1010	
[91	5B	133	0101 1011	(左方括号)
\	92	5C	134	0101 1100	(反斜线)
]	93	5D	135	0101 1101	(右方括号)
^	94	5E	136	0101 1110	(音调符号)
_	95	5F	137	0101 1111	(下划线)
`	96	60	140	0110 0000	(反引号)
a	97	61	141	0110 0001	
b	98	62	142	0110 0010	
c	99	63	143	0110 0011	
d	100	64	144	0110 0100	
e	101	65	145	0110 0101	
f	102	66	146	0110 0110	
g	103	67	147	0110 0111	
h	104	68	150	0110 1000	
i	105	69	151	0110 1001	
j	106	6A	152	0110 1010	
k	107	6B	153	0110 1011	
l	108	6C	154	0110 1100	

(续表)

字符	十进制	十六进制	八进制	二进制	
m	109	6D	155	0110 1101	
n	110	6E	156	0110 1110	
o	111	6F	157	0110 1111	
p	112	70	160	0111 0000	
q	113	71	161	0111 0001	
r	114	72	162	0111 0010	
s	115	73	163	0111 0011	
t	116	74	164	0111 0100	
u	117	75	165	0111 0101	
v	118	76	166	0111 0110	
w	119	77	167	0111 0111	
x	120	78	170	0111 1000	
y	121	79	171	0111 1001	
z	122	7A	172	0111 1010	
{	123	7B	173	0111 1011	(左花括号)
	124	7C	174	0111 1100	(竖线)
}	125	7D	175	0111 1101	(右花括号)
~	126	7E	176	0111 1110	(波浪号)
	127	7F	177	0111 1111	del



忘记 root 口令的处理方法

当使用自己的 Unix 系统时，您自己就是系统管理员，这意味着如果出现问题的话，没有其他人帮助您。

如果忘记了 **root**(超级用户)的口令，该怎么办呢？

下面给出在典型 Linux 系统中解决这一问题的步骤。各个系统之间在实际细节上可能有所不同，但是本书示范的方式适用于大部分现代版本的 Linux。

这里不会详细解释所有的命令，因为这是系统管理的领域，而这已经超出了本书的范围。如果有不理解的命令，可以参考系统的联机手册或者向他人请求帮助。

如果您有 **sudo** 的权限(参见第 6 章)，那么您可以使用下述命令快速地改变 **root** 口令：

```
sudo passwd root
```

如果没有 **sudo** 权限，或者如果您的系统被配置成不允许 **sudo** 改变 **root** 口令，那么您将发现改变 **root** 口令需要一些更精巧的措施。

通常的策略就是通过一个 Linux 光盘启动系统，接管计算机的管理。然后挂载硬盘上的主(根)文件系统，一旦完成这一步，就可以使用该挂载点作为文件系统的根，然后使用标准的 **passwd** 程序改变 **root** 口令。

- (1) 从 Live CD 引导 Linux。
- (2) 按<Ctrl-Alt-F1>组合键，进入命令行。
- (3) 改变到超级用户：

```
sudo su
```

- (4) 启动分区表编辑器：

```
parted
```

(如果系统没有 **parted**，则需要使用另一种分区编辑器，例如 **fdisk**、**cfdisk** 或 **sfdisk**。)

- (5) 在 **parted** 中，显示主硬盘的分区信息：

```
print
```

- (6) 记下包含 Linux 系统的硬盘的设备名称，例如 **/dev/hda** 或 **/dev/sda**。
- (7) 记下主 Linux 分区号，例如：分区号 2。

如果不能确定哪个是根分区，可以查看 **ext3**、**ext2**、**reiserfs** 或 **xfs** 类型的文件系统。如果这样的分区不止一个，则记下所有的号码。

- (8) 停止 **parted** 程序。

```
quit
```

现在应该会返回到 shell 提示。

- (9) 为硬盘上的文件系统创建一个挂载点(在这个例子中, 称之为 **harley**):

```
mkdir /mnt/harley
```

- (10) 通过使用由 **parted** 程序所获得的设备名和分区号, 挂载硬盘上的根文件系统。

例如, 如果设备名是 **/dev/hda**, 分区号是 2, 则使用的命令为:

```
mount /dev/hda2 /mnt/harley
```

如果在第 7 步中, 发现不止一个可能的分区, 则选择其中一个分区号。如果选择的分区号不正确, 再尝试另一个。

- (11) 确认已经挂载了根分区。为了测试这一点, 可以查看影子文件(**/etc/shadow**)是否在这个分区中, 影子文件中包含有所需的口令:

```
ls /mnt/harley/etc/shadow
```

如果没有口令文件, 则挂载的不是根分区。返回到第(10)步, 试着挂载另一个分区。继续这两步, 直至成功挂载了根分区。

- (12) 改变硬盘系统上的 **root** 口令。

有许多方法可以完成这一步。最简单的方法就是使用新挂载点作为文件系统的根运行 **passwd** 命令。这只需一条简单的命令:

```
chroot /mnt/harley passwd
```

这个 **chroot**(change root, 改变根)命令意味着: “临时将文件树的根改变为 **/mnt/harley**, 然后执行命令 **passwd**。”

因为已经位于超级用户模式, 所以使用 **passwd** 命令将改变 **root** 口令。而且因为文件系统的根已经临时改变为 **/mnt/harley**, 所以使用的口令文件是位于硬盘上的口令文件(**/mnt/harley/etc/shadow**)。

通过这种方式, 就能够改变硬盘上系统的 **root** 口令。

- (13) 移除 CD, 从硬盘重新启动系统, 测试并确保口令已经被正确改变。

名称含义

root

在 Unix 中, 名称 “root” 拥有 4 种不同的含义:

- 超级用户的用户标识
- 目录的名称, Unix 文件树的起始点
- 主 Unix 文件系统的名称
- 包含根文件系统的磁盘分区的名称

注意, 在本附录的描述中, 我们以 4 种不同的方式使用了单词 “root”。

也就是说, 为了使挂载点成为 Unix 文件树的 root 从而可以改变 root 口令, 我们挂载了位于 root 分区中的 root 文件系统。

时区与 24 小时制时间

Unix 和 Internet 在全世界使用,所以必须小心地表示时间,特别是在电子邮件和 Usenet 文章的标题中。

通常, Unix 和 Internet 都使用 24 小时制时钟。例如,在一封电子邮件的标题中,您可能看到的是 20:50,而不是 8:50 PM。如果不习惯使用 24 小时制时钟,则可以使用图 F-1 所示的转换信息。

24 小时制时间与 AM/PM 制时间之间的转换			
(午夜)	12:00 AM = 00:00	12:00 PM = 12:00	(中午)
	1:00 AM = 01:00	1:00 PM = 13:00	
	2:00 PM = 14:00	2:00 AM = 02:00	
	3:00 AM = 03:00	3:00 PM = 15:00	
	4:00 AM = 04:00	4:00 PM = 16:00	
	5:00 AM = 05:00	5:00 PM = 17:00	
	6:00 AM = 06:00	6:00 PM = 18:00	
	7:00 AM = 07:00	7:00 PM = 19:00	
	8:00 AM = 08:00	8:00 PM = 20:00	
	9:00 AM = 09:00	9:00 PM = 21:00	
	10:00 AM = 10:00	10:00 PM = 22:00	
	11:00 AM = 11:00	11:00 PM = 23:00	

图 F-1 24 小时制与 AM/PM 制的转换

每当程序需要知道时间、日期或者时区时,程序从 Unix 维护的设置中获取所需信息。例如,当发送电子邮件时,电子邮件程序在消息上添加日期、时间和时区。

为了确保时间和日期总是正确的,大多数 Unix 系统使用一个同步程序,使计算机的时钟与 Internet 上的一个准确时间源同步。该程序在后台运行,定期自动检查时间和日期,并根据需要进行修正^{*}。尽管时间检查是自动的,但是您还需要确定时区设置的正确。通常,这一过程是在安装 Unix 时进行的。

^{*} 时间检查使用所谓的 NTP(Network Time Protocol, 网络时间协议)系统执行。NTP 的作用就是同步计算机时钟和参考时钟。参考时钟可以位于同一个网络上,也可以位于 Internet 上。

完成这个工作的程序是 `ntpd`, `ntpd` 是一个在后台运行的守护进程。一些系统使用 `ntpd` 程序,该程序并不是守护进程。`ntpd` 在时间同步上更加出色,最终,它将取代 `ntpd`。基于这一原因,不推荐使用 `ntpd`。同样,由于 `ntpd` 的存在,也不推荐使用更古老的程序 `rdate`。

实际中，时区信息可以以 3 种不同方式表示。第一种表示时区信息的方式，是在具体的时间中包含本地时区的缩写。例如，下面是 **date** 命令(参见第 8 章)的输出，显示的时间是太平洋夏令时间(Pacific Daylight Time, PDT)上午 8:50:

```
$ date
Sun Dec 21 20:50:17 PDT 2008
```

另一种表示相同信息的方式是将时间转换成 UTC(Coordinated Universal Time, 协调世界时)，也称为 GMT(Greenwich Mean Time, 格林威治标准时间)或 UT(Universal Time, 世界时间)*。为了防止您对 UTC 不熟悉，下面对它进行解释。

一般认为 UTC 是国际标准时间。使用 Internet 的每个人最好能将 UTC 时间转换为自己的本地时间。如果不知道如何完成这一步，可以查看图 F-2 和图 F-3。

缩写	时区	与 UTC 的时差
UTC	协调世界时	0
GMT	格林威治标准时间	同 UTC
UT	世界时间	同 UTC
EST	东部标准时间	-5 小时
EDT	东部夏令时间	-4 小时
CST	中央标准时间	-6 小时
CDT	中央夏令时间	-5 小时
MST	山区标准时间	-7 小时
MDT	山区夏令时间	-6 小时
PST	太平洋标准时间	-8 小时
PDT	太平洋夏令时间	-7 小时

图 F-2 美国时区与 UTC 之间的关系

美国的大部分地区在三月的第二个星期日改成夏令制时间，在十一月的第一个星期日改回标准时间。

电子邮件或 Usenet 文章中通常会出现 UTC/GMT/UT 时间，即使消息或文章不是出于 UTC 时区。时间的转换由软件自动完成。作为示例，下面的时间和前面用 PDT 指定的时间相同。为了以这种格式显示当前时间，需要使用带-u 选项的 **date** 命令：

```
$ date -u
Sun Dec 21 03:50:17 UTC 2008
```

注意在这个例子中，UTC 时间是一天后的上午 3:50。这是因为 UTC 领先 PDT 时间 7 小时。出于参考目的，图 F-1 列举了美国使用的时区以及与 UTC 时间之间的时差。图 F-2 列举了欧洲和印度的时间**。

* UTC、GMT 和 UT 之间存在技术上的区别。但是，从实际使用上讲，可以认为它们是相同的。
** 时区信息实际上要比想象的更加复杂。这些表中的信息仅涵盖了一小部分世界时区。
为了记录所有这样的数据，Unix 系统使用了一个称为 **tz** 或 **zoneinfo** 数据库的标准参考库。有时候将这个数据库称为 Olson 数据库，这一名称根据美国程序员 Arthur David Olson 命名，Olson 从 20 世纪 80 年代中期开始汇编时区信息。

缩写	时区	与 UTC 的时差
UTC	协调世界时	0
GMT	格林威治标准时间	同 UTC
UT	世界时间	同 UTC
WET	西欧时间	同 UTC
WEST	西欧夏季时间	+1 小时
BST	英国夏季时间	+1 小时
IST	爱尔兰夏季时间	+1 小时
CET	中部欧洲时间	+1 小时
CEST	中部欧洲夏季时间	+2 小时
EET	东部欧洲时间	+2 小时
EEST	东部欧洲夏季时间	+3 小时
IST	印度标准时间	+5.5 小时

图 F-3 欧洲和印度时区与 UTC 之间的关系

在欧洲大多数地方，一般在三月的最后一个星期日改成夏季时间，在十月的最后一个星期日改回常规时间。

最后一种表示时间的方式就是以本地时间，后面跟着该时间与 UTC 时间之间的时差。下面举例说明，这个例子取自一封电子邮件的标题。该时间的格式与 **date** 命令显示时间的格式稍微有所不同。

Date: Sun, 21 Dec 2008 20:50:17 -0700

这个标题行显示相同的时间，即下午 8:50，并且表示本地时区与 UTC 之间有-7 小时的时差。

一旦理解了时区与 UTC 之间的区别，就可以使用图 F-2 和图 F-3 中的表转换本地时间并计算时区差。下面举例说明。

您居住在纽约，并且现在是夏天。您获得一封时间为 17:55 UTC 的电子邮件。那么用本地时间表示，您收到的电子邮件应该是什么时间呢？

在夏天，纽约使用 EDT(Eastern Daylight Time, 东部夏令时间)。通过查看表 F-2，知道 UTC 和 EDT 之间的时差为-4 小时。因此，17:55 UTC 就是 13:55 EDT，或者是纽约时间下午 1:55。

您住在加利福尼亚，在德国有一位朋友。那么这位朋友领先您多少小时？

假定现在是冬季。加利福尼亚地区使用 PST(Pacific Standard Time, 太平洋标准时间)，而德国使用 CET(Central European Time, 中部欧洲时间)。从表 F-2 和 F-3 中，我们知道 UTC 和 PST 之间的时差为-8 小时。UTC 和 CET 之间的时差为+1 小时。因此，您的朋友所在时区领先您 9 个小时。

在夏天，PST 时间变为 PDT 时间，CET 时间变为 CEST 时间。但是，两个时区的时差

没有改变，您的朋友仍然领先您 9 个小时。

您在一家技术先进的 Internet 公司工作，这家公司位于美国的西海岸。您是 Foobar 部门的部门经理，您需要安排每周一次的电话会议。参与人员中，有些程序员工作在印度的 Bangalore，而 Information Confusion 学会的副会长住在纽约。那么开会的最佳时间是什么时候呢？

首先，假定现在的时间是冬季。美国西海岸使用 PST 时间，纽约使用 EST 时间，印度使用 IST(India Standard Time, 印度标准时间)。根据上述各表，我们知道 PST 和 UTC 之间的时差为-8 小时，EST 和 UTC 之间的时差为-5 小时，IST 和 UTC 之间的时差为+5.5 小时。这意味着印度超前纽约 10.5 个小时，超前美国西海岸 13.5 小时。例如，西海岸的午夜正好是印度的下午 1:30。

假设您可以劝说程序员早点起来上班，在当地时间上午 7:30 会谈。在纽约，这将是前一天的下午 9:00。在西海岸，这是前一天的下午 6:00。这是一个不错的折衷，因此您可以安排周例会在西海岸星期一的下午 6:00，纽约星期一的下午 9:00，印度星期二的上午 7:30 召开。

在夏天，PST 时间要超前 PDT 时间 1 个小时，EST 时间要超前 EDT 时间 1 个小时。印度时间没有变化。通过查看上述各表，并重新计算，我们知道，在夏天，印度只超前纽约 9.5 个小时，超前西海岸 12.5 个小时。因此，如果保持西海岸和纽约时间不变的话，那么印度的程序员就可以等到上午 8:30，从而可以多睡 1 个小时。

名称含义

UTC、UT、GMT

格林威治(Greenwich, 发音为“Gren-itch”)是伦敦的一个区，1675 年至 1985 年间曾是英国皇家天文台所在地。正是这家天文台开发了现代的计时系统和经纬度。基于这一原因，将通过该天文台的假设南北线指定为经度 0 度。

1884 年，格林威治时间采纳为全球标准，用来确定全球各地的时区。这个全球标准时间称为格林威治标准时间(Greenwich Mean Time, GMT, 在这里单词“Mean”指平均)。GMT 在 Internet 上广泛使用，人们有时候也会采用一个更新更正式的名称来称呼它，这个名称就是世界时间(Universal Time, UT)。

除了 UT，有时候还会使用另外一种名称，即 UTC(Coordinated Universal Time, 协调世界时)。UTC 是世界时间的标准值，由美国国家标准局和美国海军天文台计算。

您可能会奇怪，为什么协调世界时的缩写是 UTC，而不是 CUT？

UTC 在 1970 年被采纳为正式的国际标准。这一工作由国际电信联盟中的一组专家完成。在命名这一新标准时，这一组专家遇到了一个问题。

在英语中，协调世界时的缩写应该是 CUT。但是在法语中，Temps Universel Coordonné 名称的缩写应该是 TUC。这些专家希望每个地方都使用相同的缩写，但是它们在使用 CUT 还是使用 TUC 上达不成一致意见。最终的妥协是使用 UTC。尽管这一缩写在英语和法语中都不正确，但是在保持和平上它拥有极大的优点。

(这一点没有多少人知道，但是现在您知道了。)

shell 选项和 shell 变量

每个 shell 都提供了一系列的方式来让您控制 shell 的行为。Bourne Shell 家族(Bash、Korn Shell)使用 shell 选项；C-Shell 家族(Tcsh、C-Shell)同时使用 shell 选项和 shell 变量。本附录汇总了这些选项和变量。

有关 shell 选项和 shell 变量的详细讨论，请参见第 12 章。

G.1 Bourne Shell 家族

Bourne Shell 家族使用大量的选项来控制 shell 的行为。特别地，交互式 shell 要求使用与非交互式 shell 不同的选项(实际上，正是这些选项使之成为交互式 shell)。指定这些选项的方式有两种。第一种方式是指定选项的标准方式，即在启动 shell 时，在命令行上指定选项。这一种方式适用于自动运行的非交互式 shell，和通过自己输入命令启动的交互式 shell。例如：

```
bash -vx
```

(顺便说一下，这两个选项在测试或调试 shell 脚本时特别有用。)

在交互式 shell 中——也就是说在 shell 提示处，可以根据希望打开或关闭选项。为了设置(打开)选项，可以使用 **set -o** 命令。为了清除(关闭)选项，可以使用 **set +o** 命令。当使用这些命令时，要指定选项的长名称，而不是缩写。例如：

```
set -o verbose
set +o xtrace
```

为了显示所有选项的当前状态，可以使用下述两条命令中的任意一条：

```
set -o
set +o
```

其中，**-o** 显示适合人类阅读的输出；**+o** 显示适合于脚本的输出(大家可以试一试)。

总之，Bourne Shell 家族使用大量的 shell 选项。在大多数情况下，不需要改变它们，因为默认值工作得很好。但是，出于参考目的，我们还是在图 G-1 中列出了 Bash 和 Korn Shell 的 shell 选项。并不是所有的选项都适合于所有的 shell，因此一定要查看最左边的列，这一列显示哪些 shell 支持这个 shell 选项。

shell	选项	长名称	含义
B K	-a	allexport	输出所有随后定义的变量和函数
•K		bgnice	以较低的优先级运行后台作业
B •	-B	braceexpand	启用花括号扩展
B K	-c	-	从字符串参数中读取命令
B K	-E	emacs	命令行编辑器: Emacs 模式, 关闭 vi 模式
B K	-e	errexit	如果命令失败, 终止并退出脚本
B K	-h	hashall	在查找到命令时 hash(记忆)命令的位置
B •	-H	histexpand	启用! 风格历史替换
B •		history	启用命令历史
B K	-I	ignoreeof	忽略 eof 信号^D, 使用 exit 退出 shell(参见第 7 章)
B K	-k	keyword	将所有的关键字参数放置在环境中
•K		markdirs	当通配时, 在目录名上追加/
B K	-m	monitor	作业控制: 启用
B K	-C	noclobber	不允许重定向输出来替换文件
B K	-n	noexec	调试: 读取命令, 检查语法, 但是不执行
B K	-f	noglob	禁止通配(文件名扩展)
•K		nolog	不在历史文件中保存函数定义
B K	-b	notify	作业控制: 当后台作业完成时立即进行通知
B K	-u	nounset	视使用已取消的变量为错误
B K	-t	onecmd	读取并执行一条命令, 然后退出
B •		posix	遵循 POSIX 标准
B •	-p	privileged	使用特许模式运行脚本或启动 shell
B •	-r	-	在受限制模式中启动 shell
•K	-s	-	排序定位参数
•K		trackall	别名: 为命令替换完整路径名
B K	-v	verbose	调试: 回显每条命令到 stderr(标准错误)
B K	-V	vi	命令行编辑器: vi 模式, 关闭 Emacs 模式
•K		viraw	在 vi 模式中, 键入字符时处理每个字符
B K	-x	xtrace	调试: 在命令执行时, 回显到标准错误(也就是跟踪)

图 G-1 Bourne Shell 家族: shell 选项

本表摘要汇总了 Bourne Shell 家族所提供的重要的 shell 选项。最左边一列显示哪个 shell 支持这个选项: B=Bash; K=Korn Shell。点号表示相应的 shell 不支持这个选项。注意有一些选项, 例如 **bgnice**, 有长名称, 但是没有短选项名称。更多的信息, 请参见特定 shell 的说明书页。

注意: (1)尽管 Bash 支持 **emacs** 和 **vi** 选项, 但是 Bash 不使用 **-E** 和 **-V**。(2)Korn shell 使用 **-h** 和 **-t**, 但是不支持长名称 **hashall** 和 **onecmd**。

在 Bash 中, 还有一些额外的选项来控制许多特性。为了设置及取消这些选项, 需要使用 **shopt**(shell options, shell 选项)命令。**shopt** 选项非常深奥, 以至于您不大可能需要它们。但是如果您有时间, 而且希望了解一下这些选项, 可以查看 **shopt** 的说明书页。

提示

阅读 Bash 的 **shopt** 选项的最佳位置就是 Bash 的说明书页。为了查找相关的讨论, 可以搜索 “shopt”。

G.2 C-Shell 家族

C-Shell 家族也使用选项控制其行为。这些选项在图 G-2 中列出。如果将图 G-2 中的选项和图 G-1 中的选项进行比较, 会发现大多数 C-Shell 选项同样也被 Bourne Shell 家族所使用。但是, 它们之间有一个重要的区别。

shell	选项	shell 变量	含义
CT	-c		从字符串参数中读取命令; 与-s 相对
•T	-d		从.cshdirs 文件中加载目录栈
CT	-e		如果命令失败, 终止并退出脚本
CT	-i		交互模式(影响提示、错误处理等)
CT	-n		调试: 读取命令, 检查语法, 但是不执行命令
CT	-s		从 stdin(标准输入)读取命令; 与-c 相对
CT	-t		读取并执行一条命令, 然后退出
CT	-v	verbose	调试: 只在历史替换之后显示每条命令
CT	-V	verbose	调试: 在读取.tcshrc/.cshrc 文件之前设置 verbose
CT	-x	echo	调试: 在所有替换之后回显每条命令
CT	-X	echo	调试: 在读取.tcshrc/.cshrc 文件之前设置 echo

图 G-2 C-Shell 家族: shell 选项

本表摘要汇总了 C-Shell 家族所提供的重要的 shell 选项。最左边一列显示哪个 shell 支持这个选项: C=C-Shell; T=Tcsh。点号表示相应的 shell 不支持这个选项。更多的信息, 请参见特定 shell 的说明书页。

注意: -v 和 -V 选项通过设置 shell 变量 **verbose** 发挥作用。-x 和 -X 选项通过设置 shell 变量 **echo** 发挥作用。

在 Bourne Shell 家族中, shell 的行为完全由选项控制, 这些选项有 3 种类型: 常规命令行选项(例如 -v)、和 **set +o** 命令一起使用的“长名称”选项以及只和 Bash 的 **shopt** 命令一起使用的大量额外选项。

尽管 C-Shell 家族只使用少量的命令行选项(参见图 G-2), 但是在很大程度上, C-Shell 家族 shell 的行为受 shell 变量的控制。第 12 章详细解释了这个 C-Shell 家族和 Bourne Shell 家族之间的重要区别。出于参考目的, 图 G-3 汇总了 C-Shell 和 Tcsh 使用的 shell 变量。

shell	shell 变量	含义
• T	addsuffix	自动补全: 在目录名后追加/(斜线)
• T	ampm	以 12 小时 AM/PM 格式显示时间
• T	autocorrect	拼写自动更正: 在试图补全之前调用单词拼写编辑器
• T	autoexpand	自动补全: 在试图补全之前调用扩展历史编辑器
• T	autolist	自动补全: 当补全失败时, 列举剩余的选择
• T	autologout	如果没有键入命令, 自动注销前等待的时间(分钟)
C T	cdpath	cd 、 chdir 、 popd 搜索的目录
• T	color	使 ls -F 命令使用颜色
• T	complete	自动补全: ignorecase =忽略大小写; enhance =同. -都是分隔符
• T	correct	拼写自动更正: cmd =命令; all =整行; complete =自动完成
C T	cwd	当前工作目录(相对于 owd)
C T	echo	调试: 在所有替换后回显每条命令
• T	echo_style	echo 命令: bsd (支持-n); sysv (支持\转义符); both ; none
• T	edit	命令行编辑器: 启用
C T	ignore	自动补全: 忽略后缀
C •	filec	自动补全: 启用(在 Tcsh 上总是启用的)
• T	group	当前组标识
C T	hardpaths	目录栈: 分解路径名, 从而使其不包含符号链接
C T	history	命令历史: 历史列表中的行号
C T	home	home 目录
C T	ignoreeof	遇到 eof 信号(^D)时不退出 shell
• T	implitcd	键入目录名即改变到该目录
• T	inputmode	命令行编辑器: 在行头设置初始模式, insert 或者 overwrite
• T	listjobs	作业控制: 列举所有挂起的作业; long =长格式
• T	loginsh	设置指示登录 shell
C T	mail	列举文件, 查看新电子邮件
• T	matchbeep	自动补全: 发出声音; ambiguous 、 notunique 、 never 、 nomatch
C T	nobeep	自动补全: 永不发出声音
C T	noclobber	不允许重定向输出替换文件
C T	noglob	通配(文件名扩展): 禁用
C T	notify	作业控制: 当后台作业结束时立即通知
• T	owd	最近[远]的工作目录(相对于 cwd)
C T	path	搜索程序的目录

图 G-3 C-Shell 家族: shell 变量

C	T	prompt	shell 提示(通过改变这个变量实现自定义)
•	T	pushdsilent	目录栈: pushd 和 popd 不列举目录栈
•	T	pushdthome	目录栈: 没有参数的 pushd 假定的 home 目录(类似于 cd)
•	T	recexact	自动补全: 精确匹配, 即使更长的匹配存在
•	T	rmstar	强制用户在执行 rm * (移除所有文件)之前确认
•	T	rprompt	在提示过程中, 显示在屏幕右边的字符串(提示: 设置 to %/)
•	T	savdirs	目录栈: 在注销之前, 保存目录栈
C	T	savehist	命令历史: 在注销之前, 保存历史列表中的行数
C	T	shell	登录 shell 的路径名
C	T	term	正在使用的终端类型
C	T	user	当前用户标识
C	T	verbose	调试: 仅在命令替换之后回显每条命令
•	T	visiblebell	使用屏闪取代可听见的声音

图 G-3 (续)

本表摘要汇总了 C-Shell 家族所提供的控制 shell 行为的重要 shell 变量。最左边一列显示哪个 shell 支持这个选项: C=C-Shell; T=Tcsh。点号表示相应的 shell 不支持这个选项。

更多的信息, 请参见特定 shell 的说明书页。有关自动补全、拼写自动更正和命令行编辑的更多信息, 请参见 Tcsh 的说明书页。有关 shell 变量的详细讨论, 请参见第 12 章。

术 语 表

本术语表包含了书中解释的 622 个技术术语的定义。在每个定义之后，方括号中的数字表示讨论这个术语的参考章号。

A

绝对路径名(absolute pathname) 一个指定目录完整名称的路径名，路径名从根目录直到实际文件。[24]

加速键(accelerator key) 指在 GUI 中使用<Alt>键充当单击某项的快捷方式的键组合。例如，当在某个窗口时，通常可以使用加速键<Alt-F>打开 File 菜单。[6]

访问时间(access time) 与文件相关，即上一次读取文件的时间。为了显示文件的访问时间，可以使用命令 `ls` 加上 `-lu` 选项。可与修改时间(modification time)比较。[24]

账户(account) 使用 Unix 系统的权限，包括一个名副其实的用户标识和口令，以及一些对该用户标识使用系统的方式的限制。[4]

动作(action) 当使用 `find` 程序搜索文件时，用来指定如何处理搜索结果的指示。例如，动作 `-print` 将搜索结果写入到标准输出中。请参阅测试(test)和运算符(operator)。[25]

活跃窗口(active window) 指在 GUI 中当前拥有焦点的窗口。无论通过键盘键入什么内容，这些内容都作为活跃窗口中正在运行的程序的输入。[6]

管理员(admin) 同系统管理员(system administrator)。[4]

别名(alias) 指在 shell 中为命令或者一系列命令指定一个用户自定义的名称。别名最常见的一种应用就是充当一个缩写或者充当命令的一个自定义变体。[13]

分配单元(allocation unit) 与硬盘或者其他存储介质上的 Unix 文件系统相关，指的是存储分配的基本单元。单元大小依赖于磁盘的类型，通常是文件系统块大小的整数倍。可与块(block)相比较。[24]

字母数字(alphanumeric) 描述仅包含字母或者数字的数据。字母数字不包含标点符号。[13]

锚(anchor) 在正则表达式中，用来匹配字符串开头或者结尾位置的元字符(^、\$、\<、\>)。[20]

匿名管道(anonymous pipe) 同管道(pipe)。[23]

应用差分(apply a diff) 为了遵循差分中包含的指示，由另一个文件重新创建一个文

件。一般情况下,人们可以通过应用差分由一个早期版本的文件重新创建一个新版本的文件。请参阅差分(diff)和补丁(patch)。[17]

参数(argument) 指键入命令时,位于命令行上的数据项,通常位于选项之后,用来向希望运行的程序传递信息。例如,在命令 `ls -l datafile` 中, `-l` 是一个选项,而 `datafile` 就是一个参数。请参阅选项(option)。[10]

ASCII 请参阅 ASCII 码(ASCII code)。[19]

ASCII 码(ASCII code) 通常简称为 ASCII。ASCII 码指的是一种标准的字符编码系统,创建于 1967 年,后来又不断进行修改。在 ASCII 码中,字符以位组合表示。每个字符存储在一个单独的字节(8 位)中。在一个字节中,最左边的位被忽略掉;其他 7 位由 0 和 1 构成一个位组合,表示一个特定的字符。ASCII 码总共包含 128 种不同的位组合,范围由 0000000 到 1111111。出于参考目的,附录 D 中列举了完整的 ASCII 码。请参阅可显示字符(printable character)。[19] [20]

ASCII 文件(ASCII file) 同文本文件(text file)。[19]

异步进程(asynchronous process) 与最古老的 Unix shell(在作业控制之前)相关,这种进程自己独自运行,与用户的输入无关。在作业控制开发出来之后,异步进程被后台进程所取代。请参阅作业控制(job control)和后台进程(background process)。[26]

自动补全(autocomplete) 指一种在 shell 中通过自动补全单词帮助用户输入命令的特性。自动补全有许多种类型,包括:命令补全(command completion)、文件名补全(filename completion)、变量补全(variable completion)、用户标识补全(userid completion)以及主机名补全(hostname completion)。[13]

B

后门(back door) 一个秘密功能,可以被黑客利用,秘密地访问系统或者控制程序。[13]

后台进程(background process) 后台进程是这样一个进程,即 shell 不等待这个进程完成就显示下一个 shell 提示。我们可以说这样的进程运行在“后台”。可与前台进程(foreground process)相比较。[26]

向后兼容(backwards compatible) 描述支持以前程序特性的程序。例如, Tcsh 向后兼容以前的 C-Shell; Bash 和 Korn Shell 向后兼容以前的 Bourne Shell。[11]

bang 感叹号(!)字符。bang 通常用来修改当前正在处理事情的模式,例如,暂停当前的程序并向 shell 发送一条命令。[9]

bang 字符(bang character) 参见 bang。[9]

bar 一个没有意义的单词,用来表示讨论或者解释过程中一个没有命名的项。单词“bar”通常和“foo”一起使用,指两个未命名的项。习惯约定是第一个项使用“foo”,第二项使用“bar”。例如,大家可能听到有些人问这样的问题:“我有两个文件,foo 和 bar。我如何才能将 foo 中所有包含某种特定模式的行复制到 bar 的末尾?”请参阅 foo 和 foobar。[9]

基 2(base 2) 同二进制系统(binary system)。[21]

基 8(base 8) 同八进制(octal)。[21]

基 10(base 10) 同十进制系统(decimal system)。[21]

基 12(base 12) 同十二进制(duodecimal)。[21]

基 16(base 16) 同十六进制(hexadecimal)。[21]

基名(basename) 与文件名(filename)是同义词。[24]

Bash Bourne shell 家族中最重要的成员,最初由 Brian Fox 于 1987 年创建, Bash 由自由软件基金会赞助。名称 Bash 代表 “Bourne-again shell”。Bash 的应用非常广泛,几乎在所有的 Linux 系统中作为默认的 shell。Bash 程序的名称为 **bash**。请参阅 Bourne shell 家族(Bourne shell family)。[11]

基本正则表达式(basic regular expression) 有时候简写为 **BRE**。指一种老类型的正则表达式,已经使用了许多年,现在被扩展正则表达式所替代。与更现代的扩展正则表达式相比,基本正则表达式,即 BRE 功能欠强大,并且语法有点模糊。基于这些原因,可以认为基本正则表达式已经过时,保留它们只是为了与一些以前的程序兼容。可与扩展正则表达式(extended regular expression)相比较。另请参阅正则表达式(regular expression)。[20]

二进制数字(binary digit) 二进制系统中使用的两个数字 0 和 1 中的一个。请参阅位(bit)和二进制系统(binary system)。[21]

二进制文件(binary file) 一种包含非文本数据的文件,只有通过程序读取,二进制文件才有意义。二进制文件的常见例子有可执行程序、对象文件、图像、音乐文件、视频文件、字处理文档、电子表格和数据库等。与文本(text)文件比较。[19]

二进制数(binary number) 通过二进制系统表示的数字,二进制系统采用基 2 编码。请参阅二进制系统(binary system)。[21]

二进制系统(binary system) 同基 2(base 2)。一种基于 2 的乘幂的计数系统,在该系统中,数字由两个数字 0 和 1 构成。请参阅八进制(octal)、十进制系统(decimal system)、十二进制(duodecimal)和十六进制(hexadecimal)。[21]

位(bit) 数据存储的基本单元,位包含一个单独的单元,且总是位于两种状态中的一种。习惯称位包含一个 0 或者包含一个 1。包含 0 的位称为“关闭”,包含 1 的位称为“打开”。术语位是“二进制数字”的缩写。请参阅字节(byte)、二进制系统(binary system)和二进制数字(binary digit)。[21]

位桶(bit bucket) 位桶指一些文件的古怪名称,这些文件拥有下述特征,即所有的输出在写入这些文件后,就被直接抛弃。位桶有两种: null 文件(/dev/null)和 zero 文件(/dev/zero)。两个文件都是伪文件,即一种特殊文件。请参阅 null 文件(null file)、zero 文件(zero file)和伪文件(pseudo-file)。[15]

块(block) 与 Unix 文件系统相关,它是文件系统本身中基本的存储分配单元,通常是 512 字节、1KB、2KB 或者 4KB。典型的 Linux 文件系统使用 1KB 大小的块。可与分配单元(allocation unit)比较。[24]

块设备(block device) 指一种如磁盘一样在读取或者写入过程中,一次处理固定数量字节的设备。在命令 **ls -l** 的输出中,字符设备由符号 **b** 表示。可与字符设备(character device)比较。[24]

引导程序(boot) 每当计算机启动或者重新启动时,引导程序用来控制计算机并且执行一些所需的初始化工作。请参阅引导程序加载程序(boot loader)和引导程序设备(boot device)。[2]

引导程序设备(boot device) 该设备由引导程序加载程序控制,读取启动操作系统所需的

数据。在大多数场合中,引导程序设备是本地硬盘上的一个分区。但是,它也可以是一个网络设备、CD、闪存驱动器等。请参阅引导程序(**boot**)和引导程序加载程序(**boot loader**)。[23]

引导程序加载程序(boot loader) 一个小型的程序,在启动或者重新启动计算机时接管控制权,为启动操作系统加载足够的软件。对于双重引导或者多重引导系统,引导程序加载程序允许选择使用哪个操作系统。最常见的 Linux 引导程序加载程序有 GRUB 和 LILO。请参阅引导程序(**boot**)和引导程序设备(**boot device**)。[2]

限定(bound) 指正则表达式中的一种规范,在{ }重复运算符中使用,匹配期望的字符一个特定的次数。例如,在重复运算符{3,5}中,限定 3、5 指定匹配 3 到 5 次。请参阅重复运算符(**repetition operator**)。[20]

Bourne shell 第一个广泛使用的 Unix shell,最初由 Steven Bourne 于 1976 年开发,Steven Bourne 是贝尔实验室的一名研究人员。尽管 Bourne shell 一直在不断更新,但是今天 Bourne shell 已经很少使用,因为它缺乏现代化 shell 的先进特性。Bourne shell 程序的名称为 **sh**。请参阅 Bourne shell 家族(**Bourne shell family**)。[11]

Bourne shell 家族(Bourne shell family) 这些 shell 的主要特征就是它们都基于 Bourne shell 和 Bourne shell 编程语言。Bourne shell 家族中最重要成员是 Bash、Korn shell、Zsh、Pdksh 和 FreeBSD shell。目前 Bourne shell 本身已经很少使用。[11]

花括号扩展(brace expansion) 与 Bash、C-Shell 或者 Tcsh 相关,Bourne shell 家族允许使用花括号将一组字符串括住,用来依次匹配或者生成每个包含该字符串的文件名。请参阅路径名扩展(**pathname expansion**)。[24]

分支(branch) 在树(数据结构)中,将一个节点添加到另一个节点的路径。分支对应于图中的一条边。请参阅树(**tree**)。[9]

浏览器(browser) 一种客户端程序,例如 Internet Explorer 或者 Firefox,用来访问网络以及其他 Internet 功能,例如电子邮件、Usenet 和匿名 FTP。对于大多数人而言,浏览器充当 Internet 的主要界面。[3]

BSD 一种操作系统,由加利福尼亚大学伯克利分校开发,最初基于 AT&T 公司的 UNIX。名称 BSD 代表“Berkeley Software Distribution”。BSD 的第一个版本于 1977 年发行,后来称为 1BSD。在 20 世纪 80 年代,BSD 是 Unix 的两个主要分支之一;另一个分支是 **System V**。[2]

BSD 选项(BSD option) 与 **ps**(进程状态)命令相关,那些派生于 20 世纪 80 年代版本的 **ps** 的选项属于 BSD(Berkeley Unix)。BSD 选项前面没有破折号。可与 UNIX 选项(**UNIX option**)。[26]

内置程序(builtin) 在 shell 中,指由 shell 直接解释的命令。与内置命令(**builtin command**)和内部命令(**internal command**)相同。[13]

内置命令(builtin command) 同内置程序(**builtin**)。[13]

字节(byte) 一种数据存储单位,包含 8 个连续位。一个字节可以存储一个单独的 ASCII 字符。请参阅位(**bit**)和二进制系统(**binary system**)。[21]

C

C 排序序列(C collating sequence) 这种排序序列基于 ASCII 码,在 C(POSIX)区域设

置中使用,并根据 C 编程语言命名。在 C 排序序列中,大写字母位于小写字母之前(ABC...XYZabc...xyz)。可与字典排序序列(dictionary collating sequence)比较。另请参阅 ASCII 码(ASCII code)、排序序列(collating sequence)和区域设置(locale)。[19]

C-Shell 一种由 Bill Joy 在 1978 年开发的 shell, Bill Joy 是加利福尼亚大学伯克利分校的一名研究生。C-Shell 的发音为“see-shell”。C-Shell 编程语言基于 C 语言(所以命名为 C-Shell)。在 C-语言的时代中, C-Shell 被广泛使用,一直是最重要的 shell 之一。今天,大多数 C-Shell 用户倾向于选择功能更强大的 Tcsh。C-Shell 程序的名称是 **csh**。请参阅 **Tcsh** 和 C-Shell 家族(C-Shell family)。[11]

C-Shell 家族(C-Shell family) C-Shell 家族的主要特征是都基于 C-Shell,特别是 C-Shell 编程语言。C-Shell 家族中最广泛使用的 shell 是 Tcsh 和 C-Shell 本身。[11]

规范(canonical)

(1)在数学中,规范指表达思想的最简单、最重要的方式。[21]

(2)在计算机科学中,规范指完成事情最常见、最传统的方式。[21]

规范格式(canonical format) 与二进制数据显示或者打印相关,最常见的使用格式由每行 16 字节构成。每行的左边是十六进制表示的偏移。每行的中间是实际字节,也是以十六进制表示,而右边是相应的 ASCII 字符。命令 **hexdump -C** 以规范格式显示二进制数据。[21]

规范模式(canonical mode) 规范模式指的是一种线路规程,即作为程序输入所键入的字符并不立即发送给程序。这些字符积聚在缓冲(存储区域)中,仅当用户按下<Return>键时才发送给程序。可与原始模式(raw mode)和 cbreak 模式(cbreak mode)比较。另请参阅线路规程(line discipline)。[21]

回车(carriage return)

(1)一种用来控制输出设备操作的特殊字符,指示光标或者打印位置应该移动到行的开头。在 ASCII 码中,回车字符以十进制 13 表示,或者十六进制 0D 表示。[4]

(2)在以前的 TeleType ASR33 上,该操作是将打印头移动到行的开头。[4]

(3)在手动打字机上,指的是杠杆,用来将回车(存放打印纸的圆筒)返回到最左端的位置上。[4]

大小写敏感(case sensitive) 描述程序或者操作系统区分大写字母和小写字母。[4]

cbreak 模式(cbreak mode) 原始模式的一种变体,除了少数几个非常重要的字符(^C、^_\、^Z、^S 和 ^Q)由终端驱动器直接处理外,大多数输入直接发送给程序的一种线路规程。可与规范模式(canonical mode)和原始模式(raw mode)相比较。另请参阅线路规程(line discipline)。[21]

CDE 一种基于 Motif 窗口管理器的桌面环境,于 20 世纪 90 年代初期开发,是多家公司与 Open Group 的一个大型协作活动的成果。CDE 是 Common Desktop Environment 的缩写。[5]

字符类(character class) 与正则表达式相关,它指一个元素,以方括号开头和结尾,包含一组字符,例如[ABCDE]。字符类用来匹配这一组字符中任何一个单独的字符。请参阅预定义字符类(predefined character class)和范围(range)。[20]

字符设备(character device) 指读取或写入数据时,一次处理一个字节的设备,例如终端。在 **ls -l** 命令的输出中,字符设备以符号 **c** 指示。可与块设备(block device)比较。[24]

字符串(character string) 字符串指一个纯文本字符序列,仅包含字母、数字或者标

点符号。[12]或者严格地讲,是一列可显示的字符。[19]等同于串(string)。请参阅可显示字符(printable character)。

字符终端(character terminal) 字符终端指的是一个只显示字符(文本):字母、数字、标点符号等的终端。同基于文本的终端(text-based terminal)。[3]

孩子(child) 同子进程(child process)。[15] [26]

子目录(child directory) 同子目录(subdirectory)。[23]

子进程(child process) 一个由另一个进程启动的进程。新进程是孩子,原始进程是双亲。请参阅父进程(parent process)。[15] [26]

和弦(chording) 当使用鼠标或者其他指点设备时,在同一时间按两个或者多个按键。[6]

CLI “command line interface(命令行界面)”的缩写。它是一种基于文本的界面,在这种界面中,用户输入直接由 shell 解释的命令。可与 GUI 比较。[6]

单击(click) 当使用鼠标或者其他指点设备时,单击指按一下按键。[6]

客户(client) 一种从服务器请求服务的程序。[3]

剪贴板(clipboard) 剪贴板指在 GUI 中一个不可见的存储区域,用来存放已经复制或剪贴的数据;此类数据可以粘贴到一个窗口中。仅当被新数据取代时,剪贴板中的数据才会变化。当关闭 GUI 时(通过关机、注销或者重启),剪贴板中的数据就会丢失。[6]

关闭(close) 关闭指在 GUI 中,停止程序在窗口中的运行,导致窗口消失。[6]

关闭按钮(close button) 关闭按钮指在 GUI 中,一个小的矩形,通常位于窗口最右边的顶部,当单击这个矩形时,会关闭窗口。[6]

簇(cluster) 与分配单元(allocation unit)同义。[24]

代码(code) 与源代码(source code)同义。[2]

排序序列(collating sequence) 排序序列描述当排序时,字符放置的顺序。在现代版本的 Unix 或 Linux 中,排序序列取决于区域设置的选择。请参阅区域设置(locale)。[19]

命令补全(command completion) 一种功能,自动补全行的开头部分键入的命令。Bash 和 Tcsh 提供了命令补全功能。请参阅自动补全(autocomplete)。[13]

命令行(command line)

(1)当输入 Unix 命令时:命令行指的是在按<Return>键之前所键入的整行。[6]

(2)当使用 vi 编辑器时:命令行指的是屏幕最底端的那一行,这一行上显示的命令是那些在键入后回显的特定命令。[22]

命令行编辑(command line editing) 命令行编辑指的是在 shell 中,允许使用一系列的命令来操纵命令行中键入内容的一种强大功能,包括使用历史列表和自动补全的能力。命令行编辑在 Bash、Korn Shell 以及 Tcsh 中可用。命令行编辑有两种模式 Emacs 模式(Emacs mode)和 vi 模式(vi mode)。[13]

命令行界面(command line interface) 请参阅 CLI。[6]

命令模式(command mode) 当使用 vi 文本编辑器时,命令模式指的是一种将键入的字符解释为命令的模式。可与插入模式(insert mode)相比较。[22]

命令处理器(command processor) 指读取或者解释从终端输入或者从文件中读取的命令的程序。shell 就是一种命令处理器。[11]

命令替换(command substitution) 命令替换指在 shell 中,将一条命令的输出插入到

另一条命令中，然后执行这条命令。为了使用命令替换，需要使用反引用字符“`”括住第一条命令。[13]

命令语法(command syntax) 同语法(syntax)。[10]

注释(comment) 注释指在一个程序中被忽略掉的行。注释由程序员使用，用来在程序中插入注解，为程序员或者其他可能阅读该程序的人提供说明资料。在 shell 脚本中，注意以#(井号)开头。[14]

兼容模式(compatibility mode) 当使用 Vim 文本编辑器时，指 Vim 采取与 vi 文本编辑器尽可能相似的模式工作。[22]

条件执行(condition execution) 条件执行指在 shell 中，仅当前一条命令(由用户指定)成功或者失败时，允许执行另一条命令的功能。[15]

配置文件(configuration file) 配置文件指文件的内容在程序启动时由程序读取的文件。一般情况下，配置文件包含影响程序运算的命令或者信息。[6]

控制台(console) 指用来管理系统的终端，被认为是主机计算机的一部分。[3]

上下文菜单(context menu) 指在 GUI 中一个包含一系列与对象相关的动作的弹出式菜单。一般情况下，通过单击鼠标右键就可以使弹出式菜单出现。[6]

成熟模式(cooked mode) 同规范模式(canonical mode)。[21]

复制(copy) 指在 GUI 中，在不修改原始数据的情况下，将数据从一个窗口复制到剪贴板中。[6]

非盈利版权(copyleft) 一种法律原则，最初适用于自由软件，授予任何人运行程序、复制程序、修改程序及发行修改后的程序(只要他们不添加自己的限制)的权限。非盈利版权的第一个实现是 GPL(GNU General Public License, GNU 通用公共许可证)。[2]

磁芯(core)

(1)最初指用在磁芯存储器中的一个微小、圆形、中空的磁性设备。[7] [21]

(2)更普遍地讲，磁芯与计算机内存同义。[7] [21]

磁芯转储(core dump) 指在 20 世纪 60 年代与 70 年代，打印输出已终止程序所使用内存的内容。磁芯转储可能要使用许多纸张，而且还需要出色的技能来解释。[7]

磁芯文件(core file) 指一个在程序终止时自动生成的文件(命名为 core)。磁芯文件包含程序终止时内存内容的副本，程序员可以分析这个文件，推断出了什么问题。[7] [21]

磁芯存储器(core memory) 一种已经废弃的计算机存储器类型，于 1952 年首次引入，由大量微小、圆形、中空的磁性设备(磁芯)组成，这些磁芯布置在点阵中，并且每个磁芯都有几条电线穿过。通过改变电线中的电流，可以改变单个磁芯的磁性，使其成为“off”，或“on”状态，从而允许存储和检索二进制数据。[7] [21]

CPU 即处理器，是计算机的主要部件。例如，程序使用的处理器时间量称为 CPU 时间。在大型机的初期，术语 CPU 代表“中央处理单元”。[8]

CPU 时间(CPU time) 进程已经在处理器上执行的时间量。请参阅时间片(slice)。[26]

骇客(cracker) 故意试图闯入计算机系统内部的人，他们总是希望做计算机管理员不允许做的事情。[4]

CSV(comma-separated value, 逗号分隔值)格式 描述包含机器可读数据的文件，在这样的文件中，各个字段通过逗号分隔，也就是说，在这样的文件中，定界符是逗号。请

参阅定界符(**delimiters**)、记录(**record**)和字段(**field**)。[17]

当前字符(current character) 当使用 **vi** 文本编辑器时, 当前光标所在位置处的字符就是当前字符。许多 **vi** 命令对当前字符执行动作。请参阅当前行(**current line**)。[22]

当前目录(current directory) 工作目录(**working directory**)的同义词。当输入 Unix 命令时, 所使用的默认目录。当前目录可由 **cd**(change directory, 改变目录)命令设置; 当前目录的名称由 **pwd**(print working directory, 显示工作目录)命令显示。[24]

当前文件(current file) 当使用分页程序 **less** 时, 当前正在查看的文件。[21]

当前作业(current job) 与作业控制相关, 指最近被挂起的作用, 或者如果没有挂起的作业, 则指最近移到后台的作业。特定的作业控制命令默认作用于当前作业。请参阅作业(**job**)、作业控制(**job control**)和上一作业(**previous job**)。[26]

当前行(current line) 当使用 **vi** 文本编辑器时, 光标当前所在的行。许多 **vi** 命令对当前行执行动作。请参阅当前字符(**current character**)。[22]

剪切(cut) 在 GUI 中, 用来从一个窗口向剪切板复制数据。作为操作的一部分, 原始数据被从原窗口删除。[6]

D

守护进程(daemon) 在后台运行的程序, 完全不与任何终端连接, 用来提供服务(在 Microsoft Windows 中, 相同类型的功能由所谓的“服务”提供)。[26]

数据结构(data structure) 指在计算机科学中, 用来组织数据的任何明确定义的方法, 利用数据结构的定义, 能够使用算法对数据进行排序、检索、修改及搜索操作。最常见的数据结构类型有列表、链接列表、关联数组、哈希表、栈、队列、双端队列(double-ended queue)以及各种基于树的结构。请参阅队列(**queue**)、栈(**stack**)和树(**tree**)。[8]

十进制数(decimal number) 一种使用十进制系统表示的数。请参阅十进制系统(**decimal system**)。[21]

十进制系统(decimal system) 同基 10。一种基于 10 的幂的计数系统, 在这种计数系统中, 数由 10 个数字 0、1、2、3、4、5、6、7、8 和 9 构成。请参阅二进制系统(**binary system**)、八进制(**octal**)和十六进制(**hexadecimal**)。[21]

默认(default) 当没有指定特定数据项时, 所使用的假定值。[10]

del 一个字符, 在一些 Unix 系统上, 用来取代退格(**^H**), 可以删除一个单独的字符。有时候 **del** 字符用两个字符 **^?** 表示。在 ASCII 码中, **del** 字符的值为八进制 127 或十六进制 7F。[7]

定界符(delimiter) 在包含机器可读数据的文件中, 用来分隔相邻字段的指定字符。以逗号作为定界符的数据是 CSV(逗号分隔值)格式。请参阅 CSV 格式(**CSV format**)、字段(**field**)和记录(**record**)。[17]

桌面(desktop) 在 GUI 中, 能够在其中工作的整个可视空间。更一般地讲, 桌面即可以在其中组织自己工作的抽象环境。桌面环境, 例如 KDE 和 Gnome, 允许使用多个桌面, 但是在某一时刻只有一个桌面可见(在 Gnome 中, 桌面被称为“工作空间”)。[6]

桌面环境(desktop environment) 指一种基于 GUI 的系统, 能够提供工作环境, 帮助

用户执行与计算机相关的复杂任务。有时候也将桌面环境称为桌面管理器。在 Linux 世界中，两个最广泛使用的桌面环境是 KDE 和 Gnome。请参阅 **KDE** 和 **Gnome**。[5]

桌面管理器(desktop manager) 同桌面环境(**desktop environment**)。[5]

破坏性退格(destructive backspace) 当光标向后移动并删除字符时所发生的退格类型。这就是按下<Backspace>键时发生的事情。可以与非破坏性退格(**non-destructive backspace**)比较。[7]

设备驱动程序(device driver) 一种充当操作系统和特定类型设备(通常是某种类型的硬件)之间界面的程序。同驱动程序(**driver**)。[2] [21]

设备文件(device file) 同特殊文件(**special file**)。[23]

字典排序序列(dictionary collating sequence) **en_US** 区域设置使用的排序序列，在这种序列中，大写字母和小写字母成对组合(**AaBbCcDd... Zz**)。可与 C 排序序列(**C collating sequence**)比较。另请参阅排序序列(**collating sequence**)和区域设置(**locale**)。[19]

字典文件(dictionary file) 一个包含在 Unix 中的文件，其中有大量的英语单词，包括大多数简明字典中的常见单词。每个单词单独一行，而且各行按字母顺序排列，从而使该文件易于搜索。任何用户都可以使用字典文件，**look** 命令和 **spell** 命令(已经废弃)也使用这个文件。[20]

死亡(die) 与进程相关，指进程停止运行。同终止(**terminate**)。[26]

差分(diff) 一串简单的编辑指示，当遵循这些指示时，可以将文件改变成另一个文件。请参阅应用(**apply**)和补丁(**patch**)。[17]

目录(directory) 3 种 Unix 文件中的一种。目录是一种位于存储设备上的文件，用来组织和访问其他文件。从概念上讲，目录“包含”其他文件。可与普通文件(**ordinary file**)和伪文件(**pseudo file**)比较。另请参阅文件(**file**)。[23]

目录节点(Directory Node) 在 Info 系统中，包含一串指向主要话题的链接的特殊节点。目录节点充当整个 Info 系统的主菜单。[9]

目录栈(directory stack) 一个由 shell 维护的栈，用来存储目录名称，可以用来改变工作目录。请参阅栈(**stack**)。[24]

发行版(distribution) 一种特定版本的 Linux 系统。[2]

distro 发行版(**distribution**)的缩写。[2]

点文件(dotfile) 一种名称以.(句点)字符开头的文件。当使用 **ls** 命令列举文件名时，除非使用 **-a**(all files, 全部文件)选项专门要求，否则不会列举点文件。同隐藏文件(**hidden file**)。[14] [24]

双击(double-click) 当使用鼠标或其他指点设备时，快速按下按键两次。[6]

拖动(drag) 在 GUI 中，用来移动图形对象。这样做时，需要使用鼠标指向对象。然后按住鼠标按键，移动鼠标(将对象移动到新位置)，然后再释放按键。[6]

驱动程序(driver) 同设备驱动程序(**device driver**)。[2] [21]

双重引导(dual boot) 计算机被设置成从两个不同操作系统中的一个引导，用户可以在启动过程中选择启动哪个操作系统。[2]

转储(dump) 同磁芯转储(**core dump**)。[21]

十二进制(duodecimal) 同基 12。一种基于 12 的幂的计数系统，在这种计数系统中，数由 12 个数字 0、1、2、3、4、5、6、7、8、9、A 和 B 构成。请参阅二进制系统(**binary system**)、八进制(**octal**)、十进制系统(**decimal system**)和十六进制(**hexadecimal**)。[21]

E

回显(echo) 在显示器上显示对应于用户刚按下的键的字符。例如, 当按下<A>键时, Unix 回显字母“A”。[3]

编辑(edit) (动词)使用文本编辑器修改文件的内容。[22]

编辑缓冲区(editing buffer) 当使用 vi 文本编辑器时, 一个包含当前正在编辑的数据的存储区域。[22]

编辑器(editor) 同文本编辑器(text editor)。[22]

Emacs 模式(Emacs mode) 在 shell 中, 命令行编辑使用的一种模式, 在这种模式中, 编辑命令与 Emacs 文本编辑器中使用的命令相同。请参阅命令行编辑(command line editing)和 vi 模式(vi mode)。[13]

仿真(emulate) 使计算机充当一种不同的设备。在使用计算机连接 Unix 主机时, 可以运行一个程序(例如 ssh)仿真终端。[3]

环境(environment) 当使用 shell 时, 复制一系列变量, 并使这些变量对所有子进程可用, 也就是说, 对任何由该 shell 启动的程序可用。[12]

环境文件(environment file) 一种初始化文件, 每当新 shell 启动时就会运行。请参阅登录文件(login file)、初始化文件(initialization file)和注销文件(logout file)。[14]

环境变量(environment variable) 在 shell 中, 存储在环境中的变量。因为环境被所有的子进程继承, 所以可以认为环境变量是全局变量。但是, 环境变量并不是严格意义上的全局变量, 因为子进程对环境变量的改变并不会传递回父进程。请参阅 shell 变量(shell variable)、全局变量(global variable)和局部变量(local variable)。[7][12]

转义(escape)

(1)当程序位于某种具体模式中时, 转义可以使程序切换到另一种不同模式。例如, 在 vi 编辑器中, 按下<Esc>键可以从插入模式转到命令模式。[13]

(2)在 shell 中, 是引用(quote)的同义词。例如, 在命令 echo hello\; goodbye 中, 我们称反斜线转义了分号。[13]

转义字符(escape character) 转义字符指一个键, 在使用程序的过程中, 当按下这个键时, 可以使程序由一种模式切换到另一种模式。[13]

事件(event) 在 shell 中, 存储在历史列表中的命令。[13]

事件号(event number) 在 shell 中, 一个用来标识事件(存入在历史列表中的命令)的数字。[13]

exec 与进程相关, 改变进程正在运行的程序。请参阅分叉(fork)、等待(wait)和退出(exit)。[26]

执行(execute) 遵循程序中包含的指令。同运行(run)。[2]

执行权限(execute permission) 一种权限类型, 允许执行文件或搜索目录。可与读权限(read permission)和写权限(write permission)比较。另请参阅文件权限(file permission)。[25]

退出(exit) 与进程相关, 停止进程的运行。请参阅分叉(fork)、exec 和等待(wait)。[26]

导出(export) 在 Bourne shell 家族(Bash、Korn shell)中, 使变量成为环境的一部分,

从而使变量可以被子进程访问。[12]

扩展正则表达式(extended regular expression) 有时候缩写为 ERE。一种正则表达式，能比旧的、传统的 Unix 正则表达式(基本正则表达式)提供功能更强大的特性。ERE 是目前的标准，属于 IEEE 1003.2(POSIX)规范。可与基本正则表达式(basic regular expression)比较。另请参阅正则表达式(regular expression)。[20]

扩展名(extension) 文件名的一个可选部分，位于文件名的末尾，跟在一个.(句点)字符的后面。例如，文件名 **foobar.c** 的扩展名为 **c**。扩展名使用户，或者程序可以标识文件的类型。[25]

外部命令(external command) 指在 shell 中，需要通过运行单独程序解释的命令。可与内部命令(internal command)比较。[13]

F

字段(field) 在包含机器可读数据的文件中，字段指记录的一个具体部分。请参阅记录(record)。[17]

FIFO

(1) “first-in, first-out, 先进先出”的缩写。描述一类数据结构，例如队列，在这类数据结构中，各个元素按照进入的顺序被读出。可与 LIFO 相比较。另请参阅队列(queue)和栈(stack)。[23]

(2)命名管道(named pipe)的同义词，发音为“fie-foe”。[23]

文件(file)

(1)文件指任何有一个名称，可以从中读取数据的源；文件还指任何一个有名称，可以向其中写入数据的目标。文件有 3 种类型：普通文件(ordinary file)、目录(directory)和伪文件(pseudo file)。[23]

(2)普通文件的同义词。[23]

文件描述符(file descriptor) 在 Unix 进程中，一个用来标识输入源或输出目标的唯一数字。默认情况下，标准输入使用文件描述符 **0**；标准输出使用 **1**；标准错误使用 **2**。[15]

文件管理器(file manager) 一个帮助用户管理文件和目录的程序。使用文件管理器是另外一种在命令行上键入目录和文件命令的方式。KDE 桌面环境的默认文件管理器是 Konqueror；Gnome 桌面环境的默认文件管理器是 Nautilus。[24]

文件模式(file mode) 一个 3 位的八进制数字，例如 755，描述 3 组文件权限：读权限(read permission)、写权限(write permission)和执行权限(execute permission)。第一个数字描述所有者(owner)的权限。第二个数字描述组(group)的权限。第三个数字描述所有用户标识的权限。[25]

文件权限(file permission) 3 种授权类型(读、写和执行)的一种，指定如何访问文件。请参阅读权限(read permission)、写权限(write permission)以及执行权限(execute permission)。[25]

文件名(filename) 路径名的最后一部分，文件的实际名称。[24]

文件名补全(filename completion) 一种自动补全, 补全部分键入的文件名。Bash、Korn Shell、C-Shell 和 Tcsh 提供有文件名补全。请参阅自动补全(**autocompletion**)。[13]

文件名生成(filename generation) 在 Korn Shell 和 Bourne Shell 中, 实现通配的功能, 也就是指通过匹配文件名替换通配符模式。在 Bash 中, 这个功能称为路径名扩展; 在 C-Shell 中和 Tcsh 中, 这一功能称为文件名替换。请参阅通配符(**wildcard**)和通配(**globbing**)。[24]

文件名替换(filename substitution) 在 C-Shell 或 Tcsh 中, 该功能实现通配, 也就是说, 通过匹配文件名替换通配符模式。在 Bash 中, 这一功能称为路径名扩展; 在 Korn Shell 和 Bourne Shell 中, 这一功能称为文件名生成。请参阅通配符(**wildcard**)、通配(**globbing**)和花括号扩展(**brace expansion**)。[24]

文件系统(filesystem)

(1)Unix 文件系统: 一种基于单一主目录(根目录)的树型层次结构, 包含 Unix 系统中的所有文件, 其中有基于磁盘文件系统、网络文件系统以及特殊用途文件系统的文件。请参阅根目录(**root directory**)和树(**tree**)。[23]

(2)设备文件系统: 一种树型层次结构, 包含存储在具体设备或磁盘分区上的文件。[23]

文件系统层次结构标准(filesystem hierarchy standard) 一种描述 Unix 系统如何组织目录, 特别是顶端目录和二级目录的标准。缩写为 FHS。[23]

过滤器(filter) 指读取和写入文本数据的任何程序, 该程序每次一行, 从标准输入读数据, 向标准输出写数据。通常, 大多数过滤器被设计为出色地完成一件事情的工具。[16]

固定介质(fixed media) 描述永久依附于计算机的存储设备, 例如硬盘。可与可移动介质(**removable media**)比较。

标志(flag)

(1)选项(**option**)的同义词。[10]

(2)当使用带**-F**选项的 **ls**(list files, 列举文件)程序时, 在文件名末尾显示的一个单独字符, 指示文件的类型。这些标志包括 **/**(目录)、*****(可执行文件)、**@**(符号连接)和 **|**(命名管道/**FIFO**)。没有标志时表示是一个普通非执行文件。[24]

焦点(focus) 在 GUI 中, 表示哪个窗口活跃的指示信息。一旦窗口拥有焦点, 那么无论在键盘上键入什么内容, 这些内容都会用作该窗口中正在运行程序的输入。[6]

同等(fold)

(1)作为形容词, 描述将小写字母视为大写字母的思想, 反之亦然。例如, “**sort** 命令有一个同等选项**-f**。” [19]

(2)作为动词, 指将小写字母视为大写字母的动作, 反之亦然。例如, “**-f** 选项告诉 **sort** 将小写字母同等于大写字母。” [19]

文件夹(folder) 当使用基于 GUI 的工具时, 文件夹是目录(**directory**)的同义词, 否则, 不要在 Unix 中使用(除非您希望听起来像一个无能的傻瓜一样)。[23]

遵循(follow) 当使用符号连接时, 遵循指引用包含在链接中的目录的名称。请参阅符号连接(**symbolic link**)。[25]

foo 一个没有意义的单词, 用来表示讨论或展示过程中没有命名的项。当必须讨论第二个没有命名的项时, 通常称之为 “**bar**”。例如, 您可能听到有人问这样的问题: “我有两个文件, **foo** 和 **bar**。如何将 **foo** 中包含某种特定模式的所有行复制到 **bar** 的末尾?” 请参

阅 **bar** 和 **foobar**。[9]

foobar 一个没有意义的单词，用来表示讨论或展示过程中没有命名的项。“foobar”通常用来表示某些类型的模式。例如，您可能在 Usenet 讨论组中看到类似于下面的问题：“我该如何移除命名为“-foobar”的文件呢？”请参阅 **foo** 和 **bar**。[9]

前台进程(foreground process) 一种必须在 shell 显示下一个 shell 提示前结束的进程。我们说这样的进程运行在“前台”。可与后台进程(**background process**)相比较。[26]

分叉(fork) 与进程相关，用来创建进程的副本。原始进程称为父进程(**parent process**)；新进程称为子进程(**child process**)。请参阅 **exec**、等待(**wait**)和退出(**exit**)。[26]

自由软件(free software) 可以被任何人合法查看、修改、共享以及发行的软件。

自由软件基金会(Free Software Foundation) 一个国际组织，由 Richard Stallman 于 1985 年创立，该组织由一小组致力于创建和发行自由软件的程序员组成。该组织是 GNU 项目的大本营。[2]

FreeBSD shell Bourne shell 家族成员，FreeBSD 操作系统默认 shell。FreeBSD shell 程序的名称是 **sh**。请参阅 Bourne shell 家族(**Bourne shell family**)。[11]

法式间距(french spacing) 印刷时，在每个句子的末尾使用 2 个空格(而不是 1 个)。通常在等宽字体——例如最初在电传打字机中使用的字体——中使用。[18]

FSF 自由软件基金会(**Free Software Foundation**)的缩写。[2]

G

通用公共许可证(General Public License) 同 **GPL**。[2]

吉字节(gigabyte) 存储计量单位，为 $2^{30}=1073741824$ 字节。缩写为 G 或 GB。请参阅千字节(**kilobyte**)和兆字节(**megabyte**)。[24]

通配(glob) (动词)globbing(通配)的动作。[24]

全局变量(global variable) 值可以在变量创建范围之外使用的变量。请参阅局部变量(**local variable**)、shell 变量(**shell variable**)和环境变量(**environment variable**)。[12]

通配(globber) 一种操作，在该操作中，通配符模式被一串匹配的文件名替换，通常是在 shell 处理的命令中。请参阅通配符(**wildcard**)和文件名扩展(**pathname expansion**)。[24]

Gnome 一种广泛使用的免费桌面环境。Gnome 项目成立于 1997 年 8 月，由两名程序员 Miguel de Icaza 和 Federico Mena 创建。他们的目标是为 KDE 创建一种备选方法，可以在更自由的许可协议下发行。可与 **KDE** 比较。请参阅桌面环境(**desktop environment**)。[5]

GNU 自由软件基金会开发的独立于商业软件的整个类 Unix 操作系统项目的名称，该项目以自由软件发行。GNU 是一个异想天开的名称，是一个递归的单词，代表“GNU’s not Unix”。请参阅自由软件基金会(**Free Software Foundation**)和自由软件(**free software**)。[2]

GNU 宣言(GNU Manifesto) 一篇由自由软件基金会主要创始人 Richard Stallman 撰写的论文，在这篇论文中，Stallman 解释了他发起自由软件运动的原因。GNU 宣言在 *Dr. Dobbs* 博士的 *Journal of Software Tools* 的 1985 年 3 月号上首次发表。[2]

GPL General Public License(通用公共许可证)的缩写。这是一个法律协议，于 1989 年首

次被自由软件基金会使用，在自由软件上实现非盈利版权。GPL 允许任何人运行程序、复制程序、修改程序以及发行修改过的程序，只要他们不在修改后的程序上添加自己的限制。[2]

图形用户界面(graphical user interface) 请参阅 GUI。[5]

图形终端(graphics terminal) 一种不仅能够显示字符，而且还能够使用小点在屏幕上绘制任何内容，包括图形、几何图案、阴影、线、颜色等的终端。[3]

组(group)

(1)一组共享公共文件权限的用户标识。组可以使在一起工作的人方便地读、写或执行其他人的文件。组的名称称为组的组标识(groupid，发音为“group-I-D”)。请参阅组标识(groupid)、主组(primary group)、辅组(supplementary group)和文件模式(file mode)。[25]

(2)在正则表达式中，指圆括号中的一串字符，这一串字符可以视为一个单独的单元，通常位于重复运算符之前，例如(xyz){5}。在这个例子中，组(xyz)与重复运算符{5}组合使用，可以连续匹配字符串“xyz”5次。[20]

组标识(groupid) 用于共享文件权限的组的名称。发音为“group-I-D”。请参阅组(group)。[25]

编组(grouping) 在 shell 中，特别是在 C-Shell 中，一串在圆括号中键入的命令，被子 shell 执行。[15]

GUI “graphical user interface，图形用户界面”的缩写，发音或者为“gooey”，或者为3个单独的字母“G-U-I”。GUI是一种允许使用指点设备(通常是鼠标)和键盘管理窗口、图标、菜单和其他图形元素与计算机进行交互的系统。可与 CLI 比较。[5]

H

破解(hack) 通常是指通过编程，做大量探测系统的工作。[4]

黑客(hacker) 做破解的人。[4]

硬链接(hard link) 链接(link)的同义词。用来区分常规链接(硬链接)和符号链接(软链接)。[25]

硬件(hardware) 计算机的物理部件，包括：键盘、显示器、鼠标、硬盘、处理器、内存等。[2]

头文件(header file) 同包含文件(include file)。[23]

无头系统(headless system) 一种自己运行，不需要人类直接输入的计算机。典型的无头系统通常是服务器，服务器一般没有显示器、键盘或鼠标。当需要时，可以通过网络连接控制服务器。[3]

hex 同十六进制(hexadecimal)。[21]

十六进制(hexadecimal) 同基16。一种基于16的幂的计数系统，在这种计数系统中，数由16个数字0、1、2、3、4、5、6、7、8、9、A、B、C、D、E和F构成。十六进制通常缩写为 hex。请参阅二进制系统(binary system)、八进制(octal)、十进制系统(decimal system)和十二进制(duodecimal)。[21]

隐藏文件(hidden file) 一种名称以.(句点)字符开头的文件。当使用 ls 命令列举文件名

时，除非使用 **-a**(all files, 全部文件)选项明确要求，否则不会列举隐藏文件。同点文件(dotfile)。[14] [24]

历史列表(history list) 在 shell 中，一串已经输入过的命令。用户可以通过不同的方式访问历史列表——具体访问细节依赖于所使用的 shell——调用前面输入过的命令，然后可以对命令进行修改并重新输入。一些 shell 允许用户设置历史列表的大小，并指定是否在用户注销时保存列表。[13]

保持(hold) 当使用鼠标或其他指点设备时，按下按键并在执行动作(例如移动窗口)的过程中保持按住按键不放。[6]

home 目录(home directory) 设计用来存放特定用户标识的文件的目录。无论何时，当登录时，当前目录会被自动地设置为 home 目录。[23]

主机(host) 运行 Unix 的计算机。用户可以使用终端连接主机。[3]

主机名补全(hostname completion) 一种自动补全，当单词以@字符开头时，可以补全局部键入的计算机名称。需要补全的单词必须是本地网络上计算机的名称。Bash 提供有主机名补全功能。请参阅自动补全(autocompletion)。[13]

人类可读(human-readable) 程序的输出被设计为适合于人类阅读。可与机器可读(machine-readable)比较。[12]

I

图标(icon) 在 GUI 中，一个表示对象(例如窗口、程序或文件)的小图形。[6]

图标化(iconify) 同最小化(minimize)。[6]

空闲进程(idle process) 进程#0，即原始进程，在引导过程中创建。空闲进程执行许多初始化功能，创建进程#1(初始化进程)，然后运行一个非常简单的程序(该程序实质上执行一个不做任何事情的无穷循环，因此，将该进程命名为“空闲进程”)。请参阅引导程序(boot)和初始化进程(init process)。[26]

空闲时间(idle time) 当使用 CLI(命令行界面)时，用户最后一次按键过去的时间。[8]

包含文件(include file) 一种包含可以在需要时插入到程序中的 C 或 C++源代码的文件。典型的包含文件中包含有子例程、数据结构、变量、常量等的定义。请参阅头文件(header file)。[23]

索引节点(index node) 同 i 节点(inode)。[25]

索引号(index number) 同 i 节点号(inumber)。[25]

中缀表示法(infix notation) 一种算术表示法，在这种表示法中，运算符位于操作数的中间，例如“5+7”。可与前缀表示法(prefix notation)和后缀表示法(postfix notation)相比较。[8]

Info 一种帮助系统，派生于 Emacs，并且独立于 Unix 联机手册，用来文档化 GNU 实用工具。[9]

Info 文件(Info file) 在 Info 系统中，包含一个主题文档资料的文件。[9]

继承(inherit) 与子进程相关，获得父进程的环境中变量副本的访问权。[12]

初始化进程(init process) 进程#1，由进程#0(空闲进程)创建。初始化进程是系统上其他所有进程的祖先。初始化进程完成引导过程的最后一部分。在其他任务中，初始化进程以一个

特定的运行级别启动系统。初始化进程还收养所有的孤儿进程,确保孤儿进程能够得到适当处理。请参阅引导程序(**boot**)、空闲进程(**idle process**)、运行级别(**runlevel**)和孤儿(**orphan**)。[26]

初始化文件(initialization file) 一个包含当用户登录或新 shell 启动时执行的命令的文件。初始化文件有两种类型:在用户登录时运行的登录文件(**login file**)和启动新 shell 时运行的环境文件(**environment file**)。请参阅注销文件(**logout file**)。[14]

内联接(inner join) 一种联接类型,在这种类型的联接中,输出只包含那些联接字段匹配的行。可与外联接(**outer join**)比较。请参阅字段(**field**)、联接(**join**)和联接字段(**join field**)。[19]

i 节点(inode) 索引节点的缩写,发音为“eye-node”。在文件系统中,用来存放文件基本信息的结构。请参阅 i 节点表(**inode table**)、i 节点号(**inumber**)和链接(**link**)。[25]

i 节点表(inode table) 一个包含文件系统中所有 i 节点的表。每个 i 节点描述一个文件,文件由其 i 节点号标识。请参阅 i 节点(**inode**)和 i 节点号(**inumber**)。[25]

输入模式(input mode) 当使用 vi 文本编辑器时,一种将键入的字符插入到编辑缓冲区中的模式。可与命令模式(**command mode**)比较。[22]

输入流(input stream) 程序读取的数据。可与输出流(**output stream**)相比较。另请参阅流(**stream**)。[19]

交互(interactive) 用来描述与人通信的程序。当运行交互式程序时,输入来自键盘或鼠标,输出发送给显示器。[12]

交互式 shell(interactive shell) 一种为在终端工作的用户提供界面的 shell。[12]

界面(interface) 机器的一部分,为用户与机器交互提供方法。例如,对于桌面计算机而言,界面包括显示器、键盘、鼠标、扬声器,还有可能包括麦克风、网络摄像机。[3]

内部命令(internal command) 在 shell 中,能够由 shell 直接解释的命令。同内置命令(**builtin command**)和内部程序(**builtin**)。可与外部命令(**external command**)相对照。[13]

进程间通信(interprocess communication) 两个进程间数据的交换。缩写为 IPC。请参阅信号(**signal**)。[23]

i 节点号(inumber) 索引号的缩写,发音为“eye-number”。在文件系统中,用来标识 i 节点表中特定 i 节点(索引节点)的数字。请参阅 i 节点(**inode**)、i 节点表(**inode table**)和链接(**link**)。[25]

J

作业(job) 在 shell 中,当前正在运行或挂起命令的内部表示。在大多数情况中,一个作业对应于一个单独的进程。但是,在管道线或复合命令中,一个作业对应于多个进程。作业与进程不同,作业由 shell 控制,进程由内核管理。请参阅作业控制(**job control**)、作业号(**job number**)和作业表(**job table**)。可与进程(**process**)比较。[26]

作业控制(job control) 一种由内核支持,shell 实现的功能,允许用户运行多个进程,一个进程在前台,其他进程在后台运行。有了作业控制,用户可以在前台和后台之间来回移动进程、挂起(暂停)进程以及显示进程的状态。请参阅作业(**job**)、作业号(**job number**)和作业表(**job table**)。[26]

作业 ID(job ID) 同作业号(job number)。[26]

作业号(job number) 一个唯一的数字, 由 shell 分配, 用来标识一个特定的作业。有时候称为作业 ID。作业号从 1 开始, 并向上递增。请参阅作业(job)、作业控制(job control)和作业表(job table)。可与进程 ID(process ID)比较。[26]

作业表(job table) 一个由 shell 维护的表, 用来记录每个用户标识启动的全部作业。在作业表中, 每个进程包含一个条目, 条目由作业 ID 索引。表中的每个条目包含描述及管理一个特定作业所需的信息。请参阅作业(job)、作业控制(job control)和作业 ID(job ID)。可与进程表(process table)比较。[26]

联接(join) 基于匹配字段将两组数据组合起来。请参阅字段(field)、联接字段(join field)、内联接(inner join)和外联接(outer join)。[19]

联接字段(join field) 当创建联接时, 用来匹配两组数据的字段。请参阅字段(field)、联接(join)、内联接(inner join)和外联接(outer join)。[19]

K

KDE 一种广泛使用的免费桌面环境。KDE 项目成立于 1996 年 10 月, 由一名德国学生 Matthias Ettrich 创建。Ettrich 的目标就是创建一个完全集成的图形工作环境。实际上, KDE 是第一个现代的桌面环境。可与 Gnome 比较。另请参阅桌面环境(desktop environment)。[5]

内核(kernel) 操作系统的中心部分, 一直处于运行状态, 在需要时提供基本的服务。[2]

关键字(keyword) 在 shell 中, 若干个用来控制 shell 脚本中流的内置命令之一。例如, 对于 Bash 来说, 关键字有 case、for、function、if、select、time 和 while。[13]

杀死(kill) 永久地终止进程。在普通环境中, 前台进程可以通过按[^]C 或键入退出命令(在原始模式中运行的程序)杀死。可与挂起(suspend)比较。[12]

千字节(kilobyte) 一种存储计量单位, $2^{10}=1024$ 字节。缩写为 K 或 KB。请参阅吉字节(gigabyte)和兆字节(megabyte)。[24]

Korn shell Bourne shell 家族成员, 作为 Bourne shell 的替代品, 于 1982 年由 David Korn(贝尔实验室的研究员)开发。Korn shell 程序的名称是 ksh。请参阅 Pdksh 和 Bourne shell 家族(Bourne shell family)。[11]

L

滞后(lag) 当使用远程 Unix 系统时, 按下某个键或移动鼠标与观察到动作结果之间的一个显而易见的延迟。

抽象层次(layers of abstraction) 一种模型, 在这种模型中, 大型的整体目标根据层次定义, 可视化为自底向上的堆叠式结构。每个层次向其上一层提供服务, 并向其下一层请

求服务,再没有其他交互作用。[5]

叶子(leaf) 在树(数据结构)中,只有一个连接的节点,也就是说,一个位于分支末端的节点。请参阅树(tree)。[9]

左键(left button) 在鼠标或其他指点设备上,当鼠标在右手上时,最左边的按键。请参阅右键(right button)和中间键(middle button)。[6]

左单击(left-click) 在使用鼠标或其他指点设备时,按下左键。[6]

库(library) 一个已经存在的数据和代码模块,通常用来允许程序访问操作系统提供的服务。[23]

LIFO “last in, first out(后进先出)”的缩写。用来描述诸如栈之类的数据结构,在这样的数据结构中,数据按进入的相反顺序被检索。可与 **FIFO** 比较。另请参阅栈(stack)和队列(queue)。[8] [24]

线路规程(line discipline) 终端驱动程序使用的一种功能,用来为交互式界面提供所需的预处理和后处理。请参阅规范模式(canonical mode)和原始模式(raw mode)。[21]

行编辑器(line editor) 同面向行的编辑器(line-oriented editor)。[22]

面向行的编辑器(line-oriented editor) 一种对文本行编号并且使用的命令基于这些编号的文本编辑器。同行编辑器(line editor)。可与面向屏幕的编辑器(screen-oriented editor)比较。[22]

换行符(linefeed)

(1)一个用来控制输出设备操作的特殊字符,指示光标或打印位置应该移动到下一行。在 Unix 中,换行符称为新行,用来指示文本行的末尾。当发送给终端时,换行符将导致光标向下移动一行。在 ASCII 码中,换行符是 **^J**(Ctrl-J),十进制值为 10,十六进制值为 0A。请参阅新行(newline)和回车(return)。[4] [7]

(2)在旧型的 Teletype ASR33 上,打印头向下移动一行的操作。[4]

链接(link) 在 Info 系统中,从一个节点跳到另一个节点的功能。[9]

Linus 定律(Linus's Law) “Given enough eyeballs, all bugs are shallow.(让足够多的人阅读源代码,错误将无所遁形)。”换句话说,就是当有大量的人测试及阅读新代码时,不用花很长时间就会找到代码的 bug。Linus 定律在 Eric Raymond 的论文 *The Cathedral and the Bazaar* 中提起。选择这一名称是为了尊敬 Linus Torvalds,因为他是 Linux 内核的创始人。

Linux

(1)任何由 Linux 项目创建的一类 Unix 内核。Linux 项目由 Linus Torvalds 于 1991 年启动,目的是开发这样的内核。[2]

(2)更一般地讲,指任何基于 Linux 内核的操作系统。这样的操作系统有时候称为 GNU/Linux,表示 Linux 内核和 GNU 实用工具组合在一起。[2]

live CD 一种可用于启动计算机的光盘,光盘包含运行一个完整操作系统所需的全部内容。当从 live CD 启动时,就会绕过硬盘,从而可以不在自己的系统上安装该操作系统而使用它。[2]

局部(local) 用于描述变量只存在于变量创建的范围之内。例如,shell 变量在变量所在 shell 中就是局部的。[12]

局部变量(local variable) 一种只在自己创建范围之内存在的变量。例如,在 shell 中,不属于环境的变量就是局部变量。请参阅全局变量(global variable)、shell 变量(shell

variable)和环境变量(**environment variable**)。[12]

区域设置(locale) 一种描述语言和风俗习惯的规范,可以用来与特定文化的用户沟通。其目的就是使用户可以选择自己希望的区域设置,而且其所运行的程序就会相应地与他进行通信。对于美国英语文化的用户,默认区域设置或者是基于 ASCII 码的 **C(POSIX)** 区域设置;或者是 **en_US** 区域设置,该区域设置属于新的国际系统。[19]

登录(log in) 启动一个 Unix 工作会话。[4]

注销(log out) 终止一个 Unix 工作会话。[4]

登录(login) 描述登录系统的过程。[4]

登录文件(login file) 一种初始化文件,当用户登录时运行。请参阅环境文件(**environment file**)、初始化文件(**initialization file**)和注销文件(**logout file**)。[14]

登录 shell(login shell) 登录时自动启动的 shell。请参阅非登录 shell(**non-login shell**)。[11][14]

注销(logout) 描述退出系统的过程。[4]

注销文件(logout file) 包含用户注销时所执行命令的文件。请参阅初始化文件(**initialization file**)、登录文件(**login file**)和环境文件(**environment file**)。[14]

小写字母(lowercase) 描述小写字母“a”至“z”。[4]

M

机器可读(machine-readable) 将程序的输出格式化为适合另一个程序处理的格式。这样的输出可能不适合人阅读。可与人类可读(**human-readable**)比较。[12]

宏(macro) 当使用 **vi** 文本编辑器时,命令的一个单字符缩写。[22]

邮件服务器(mail server) 为各种客户端发送及接收电子邮件提供服务的计算机。[3]

大型计算机(mainframe computer) 一种大的、价格昂贵的计算机,通常由机构(例如政府、大学和公司)使用,可以为许多程序员和管理员提供服务。名称“mainframe,大型机”于 20 世纪 70 年代初期开始使用,以便区分新的小型计算机和以前旧的大型计算机。[3]

说明书页(man page) 在联机手册中,一个主题的文档资料。根据约定,每个话题的文档资料称为一个“页面”,即使这个主题可能包含许多内容,需要打印许多页。同页面(**page**)。请参阅联机手册(**online manual**)。[9]

手册(Manual) 同联机手册(**online manual**)。当 Unix 用户称“手册”时,他们通常指联机手册。[9]

映射(map) (动词)创建一个映射。[7]

已映射(mapped) 表示两个对象之间的映射已经存在。例如,<Ctrl-C>字符映射到 **intr** 信号。因此,当按<Ctrl-C>键时,它的作用就是发送 **intr** 信号。[7]

映射(mapping) (名词)两个对象之间的等价关系。例如,如果我们说 A 映射到 B,那么这意味着当我们使用 A 时,就等同于使用 B。[7]

匹配(match) 与正则表达式相关,对应于特定的字符串。[20]

最大化(maximize) 在 GUI 中,将窗口扩展成最大的尺寸。[6]

最大化按钮(maximize button) 在 GUI 中, 一个小方框, 通常位于窗口的右上角, 当单击这个按钮时, 可以最大化窗口。[6]

兆字节(megabyte) 一种存储计量单位, $2^{20}=1048576$ 字节, 缩写为 M 或 MB。请参阅**千字节(kilobyte)**和**吉字节(gigabyte)**。[24]

内存转储(memory dump) 同**磁芯转储(core dump)**。[21]

菜单(menu) 一串选择项, 可以从中选取相应项。在 GUI 中, 有两种类型的菜单: 比较常见的下拉菜单(**pull-down menus**)和不太常见的弹出式菜单(**pop-up menus**)。[6]

菜单栏(menu bar) 在 GUI 中, 一个位于窗口顶端由一系列水平单词构成的菜单。[6]

元字符(metacharacter) 在 shell 中, 指一个拥有特殊非字面含义的字符。[13]

微内核(microkernel) 一种包含相对少量的程序, 并调用其他程序(称为服务器)完成大量工作的内核。根据微内核的本质, 微内核效率有点低。但是, 因为微内核的模块结构, 它比其他内核更易于维护和定制。可与**单内核(monolithic)**比较。[2]

中间键(middle button) 指鼠标或其他指点设备上中间的按键。尽管几乎所有的鼠标都有左键和右键, 但是许多鼠标没有中间键。请参阅**右键(right button)**和**左键(left button)**。[6]

中间单击(middle-click) 指使用鼠标或其他指点设备时, 按下中间键。[6]

微型计算机(minicomputer) 指 20 世纪 70 年代和 80 年代制造的相对较小、价格不昂贵的计算机。直到个人计算机在 20 世纪 80 年代面世, 大多数 Unix 系统都运行在微型计算机上。可与**大型计算机(mainframe)**比较。[3]

最小化(minimize) 在 GUI 中, 使窗口从屏幕上的主要部分消失。为了恢复已经最小化的窗口, 可以单击这个窗口在任务栏上的图标。[6]

最小化按钮(minimize button) 位于 GUI 中的一个小小方框, 通常位于窗口的右上角, 当单击这个小小方框时, 就会最小化窗口。[6]

模式(mode)

(1)指程序或设备的一种特定状态。例如, 可以以文本模式(通过使用 CLI)或图形格式(通过使用 GUI)使用 Unix。同样, 当使用 vi 编辑器时, 在任何时间, 或者位于命令模式中, 或者位于插入模式中。[6] [13]

(2)同**文件模式(file mode)**。[25]

修改时间(modification time) 与文件相关, 指文件的上一次改变时间。为了显示文件的修改时间, 可以使用带 -l 选项的 ls 程序。可与**访问时间(access time)**比较。[24]

修饰键(modifier key) 指键盘上的键, 在按住这些键的同时可以按另一个键, 例如 <Shift>、<Ctrl>和<Alt>。[7]

单内核(monolithic kernel) 一种包含一个单一的、相对大型的程序的内核, 能够在内部执行所有操作。单内核效率较高, 但是比较笨拙, 因而使单内核的设计和维持困难。可与**微内核(microkernel)**比较。[2]

挂载(mount) (动词)通过将设备文件系统连接到主 Unix 文件系统, 允许访问设备上的文件系统。在挂载文件系统时, 需要使用 mount 命令。设备文件系统依附的目录称为**挂载点(mount point)**。请参阅**卸载(unmount)**和**文件系统(filesystem)**。[23]

挂载点(mount point) Unix 文件系统中连接(也就是挂载)设备文件系统的目录。请参阅**挂载(mount)**和**文件系统(filesystem)**。[23]

多重引导(multi-boot) 计算机被设置成可以从多个操作系统启动, 用户可以在启动过程中选择启动哪个操作系统。[2]

多道程序设计(multiprogramming) 多任务处理的旧称, 指操作系统可以同时执行多个程序。[3]

多任务处理(multitasking) 指操作系统可以同时执行多个程序。[2]

多用户(multiuser) 指操作系统可以同时支持多个用户。[2]

N

名称(name) 用来引用变量的标识符。[12]

命名管道(named pipe) 一种伪文件, 用来将程序的输出连接到第二个程序的输入。与匿名管道(常规管道)不同, 命名管道明确创建, 并且在删除之前一直存在。因此, 命名管道可以重复使用。命名管道也叫 FIFO(“fie-foe”)。可与管道线(pipeline)比较。另请参阅伪文件(pseudo file)。[23]

新行(newline) 换行符的 Unix 名称。在文本中, 新行字符用来指示一行的末尾。当发送给终端时, 新行会使光标向下移动一行。在 ASCII 码中, 新行字符是 ^J(Ctrl-J), 其值为 10(十进制)或 0A(十六进制)。[7]

News Usenet 的同义词。[3]

新闻服务器(news server) 存储 Usenet 文章的服务器, 用户通过新闻阅读程序可以阅读这些文章。[3]

新闻组(newsgroup) Usenet 讨论组。[3]

新闻阅读程序(newsreader) 一种用户访问 Usenet 讨论组系统的客户程序。[3]

nice 值(nice number) 一个用来修改进程优先级的数字。nice 值越高, 进程的优先级越低。在大多数情况下, 进程的 nice 值通过 nice 或 renice 命令改变, 以降低进程的优先级。[26]

niceness nice 值(nice number)的同义词。[26]

节点(node)

(1)在树(数据结构)中, 路径中的一个分支, 对应于图中的一个顶点。请参阅树(tree)。[9]

(2)在 Info 系统中, 指包含一个具体主题信息的一节。[9]

非破坏退格(non-destructive backspace) 指光标向后移动但不改变任何内容的退格类型。当按下 <Left> 箭头键时, 发生的就是这种情况。[7]

非交互式(non-interactive) 指程序独立于人运行。例如, 非交互式程序可以从文件读取输入, 将输出写入到另一个文件中。[12]

非交互式 shell(non-interactive shell) 正在运行 shell 脚本的 shell。[12]

非登录 shell(non-login shell) 在用户登录时不启动的交互式 shell。请参阅登录 shell(login shell)。[14]

非规范模式(noncanonical mode) 同原始模式(raw mode)。[21]

null 描述没有值的变量。[12]

null 字符(null character) 也称为 null。字符的所有位都是 0, 也就是数值 0。请参阅

zero 文件(**zero file**)。[23]

null 文件(null file) 伪文件/dev/null。当使用空 null 文件作为输出目标时,意味着将所有的输入抛弃。当将空文件用作输入源时,总是返回 **eof** 信号(也就是没有任何东西)。null 文件是两个位桶(**bit buckets**)中的一个,另一个是 zero 文件(**zero file**)。请参阅伪文件(**pseudo-file**)。[23]

编号缓冲区(numbered buffer) 当使用 vi 文本编辑器时,有 9 个存储区域(编号 1 至 9)用来存储及检索数据。请参阅无名缓冲区(**unnamed buffer**)。[22]

O

八进制(octal) 同基 8。一种基于 8 的幂的计数系统,在这种计数系统中,数由 8 个数字 0、1、2、3、4、5、6 和 7 构成。请参阅二进制系统(**binary system**)、十进制系统(**decimal system**)、十二进制(**duodecimal**)和十六进制(**hexadecimal**)。[21]

偏移(offset) 文件中特定字节的位置,第一个字节的偏移为 0。[21]

一个或多个(one or more) 表示必须至少使用一个。例如,某条命令的语法可能允许指定一个或多个文件名。这意味着必须至少使用一个文件名。可与 0 个或多个(**zero or more**)比较。[10]

联机/在线(online)

(1)按以前的意义讲,指能够连接到一台具体计算机系统。例如,当您连接到远程 Unix 主机时,就称您联机了。[9]

(2)描述 Internet 资源或服务,例如在线银行业务。[9]

(3)描述某个正在使用 Internet 的人,例如:“Linda 现在正在线。”[9]

(4)描述一种因为 Internet 才存在的状况,例如一种联机关系。[9]

联机手册(online manual) 一个信息集合,在任何时候对所有的 Unix 用户可用,包含命令和重要系统功能的文档资料。联机手册分成一节一节的,每节包含许多条目(称为页),每个条目记录一个单独的主题。联机手册可以使用 **man** 命令访问,或者在 GUI 中使用 **xman** 访问。另外,在网络上也可以找到许多版本的联机手册。Unix 中有一个很久的传统,即用户在请求帮助之前应该先查看联机手册。请参阅 **RTFM**。当 Unix 用户谈论“手册”时,他们所指的通常是联机手册。[9]

开放软件基金会(Open Software Foundation) 一个国际组织,由 8 家 Unix 厂商(包括 IBM、DEC 和 HP 公司)成立于 1988 年 5 月,目的是创建自己的“标准”Unix。开放软件基金会缩写为 OSF。可与 Unix 国际(**Unix International**)比较。[5]

开放源代码运动(open source movement) 程序员之间的一个松散组织的国际社会运动,基于一同开发自由软件的意愿。[2]

开放源代码软件(open source software)

(1)源代码自由发行的软件,源代码通常和软件一起发行。[2]

(2)自由软件(**free software**)的同义词。[2]

操作系统(operating system) 一个复杂的主控程序,其主要功能是高效地利用硬件。操作系统充当硬件对用户和程序的主要界面。[2]

运算符(operator) 当使用 **find** 程序时, 一个用来组合或修改测试的指令。例如, **!** 运算符对测试的含义求反。请参阅测试(**test**)和动作(**action**)。[25]

选项(option)

(1)当输入 Unix 命令时, 它是命令的一个可选部分, 几乎总是以-或--(一个或两个连字符)开头, 指明希望命令执行的方式。例如, 您可能输入命令 **ls -l**。这是带**-l**选项的 **ls** 命令。根据约定, -字符通常发音为“dash”, 尽管实际上它是一个连字符或负号(取决于自己的观点)。如果讨论的是上一个例子, 那么您可以说您使用了带“dash L”选项的 **ls** 命令。请参阅参数(**argument**)。[10]

(2)当使用 **vi** 文本编辑器时, 它是允许控制程序特定方面的操作的设置。**vi** 选项有两种类型: 开关(**switch**)和变量(**variable**)。[22]

普通文件(ordinary file) 3 种 Unix 文件的一种, 同常规文件(**regular file**)。指的是包含有数据并位于硬盘、CD、DVD、闪存、内存卡或软盘等存储设备上的文件。就此而言, 普通文件是最常使用的文件类型。可与目录(**directory**)和伪文件(**pseudo file**)比较。另请参阅文件(**file**)。[23]

孤儿(orphan) 父进程已经终止的子进程。请参阅死亡(**die**)和僵尸进程(**zombie**)。在许多系统上, 孤儿被进程#1, 即初始化进程收养, 从而使孤儿进程可以被适当处理。请参阅初始化进程(**init process**)。[26]

OSF 开放软件基金会(**Open Software Foundation**)的缩写。[5]

外联接(outer join) 一种联接, 在这种联接中, 输出还包含联接字段不匹配的行。可与内联接(**inner join**)比较。请参阅字段(**field**)、联接(**join**)和联接字段(**join field**)。[19]

输出流(output stream) 由程序写入的数据。可与输入流(**input stream**)比较。请参阅流(**stream**)。[19]

所有者(owner) 与文件相关, 即控制文件权限的用户标识。默认情况下, 创建文件的用户标识是文件的所有者。但是, 可以使用 **chown** 程序改变文件的所有者。[25]

P

页面(page) 在联机手册中, 一个单独主题的文档资料。根据传统, 每个主题的文档资料称为一个“页面”, 尽管主题可能有几张打印页面那么多。同说明书页(**man page**)。请参阅联机手册(**online manual**)。[9]

分页程序(pager) 一种每次一屏地显示文件或管道线中文本的程序。[21]

段落(paragraph) 当使用 **vi** 文本编辑器时, 一段以空行开头及结尾的文本。当移动光标或修改文本时, 有不同的 **vi** 命令对段落起作用。请参阅单词(**word**)和句子(**sentence**)。[22]

双亲(parent) 同父进程(**parent process**)。[15] [26]

父目录(parent directory) 一个包含另一个目录的目录。位于父目录中的目录称为子目录(**subdirectory** 或 **child directory**)。请参阅目录(**directory**)。[23]

父进程(parent process) 启动另一个进程的进程。原始进程称为双亲, 新进程称为孩子。请参阅子进程(**child process**)。[15] [26]

解析(parse) 在 shell 中, 将命令分隔成逻辑部件, 然后再对逻辑部件进行分析和解释。[13]

偏序(partial ordering) 当一组元素排序时, 二元关系只针对组内一部分成员, 而不是全部成员。可与全序(total ordering)比较。[19]

分区(partition) 指在诸如硬盘之类的存储设备上, 一个逻辑分离的硬盘, 在这块分区上可以安装操作系统或文件系统。[2]

分区管理器(partition manager) 一个用来管理磁盘或相似设备上分区的程序。[2]

口令(password) 一种秘密模式的字符, 必须作为登录过程的一部分键入, 从而确保用户有权使用特定的用户标识。[4]

口令过期时间(password aging) 一种强制用户定期改变口令的安全要求, 例如每隔 60 天改变一次口令。[4]

口令文件(password file) 一个系统文件, 即/etc/passwd, 包含系统中所有用户标识的信息。文件中的每行包含有一个用户标识的信息。在旧系统上, 口令文件包含有口令(当然是编码的)。在现代系统上, 口令单独保存在影子文件中。请参阅影子文件(shadow file)。[11]

粘贴(paste) 在 GUI 中, 将数据从剪贴板复制到窗口中。剪贴板中的数据不会发生任何变化。[6]

补丁(patch) 用来修改程序的差分, 通常以某种方式修复 bug, 或增加程序。请参阅应用(apply)和差分(diff)。[17]

路径(path) 同路径名(pathname)。[24]

路径名(pathname) 目录树中文件位置的描述, 同路径(path)。请参阅绝对路径名(absolute pathname)和相对路径名(relative pathname)。[24]

路径名扩展(pathname expansion) 在 Bash shell 中, 用来实现通配的功能, 也就是说通过匹配文件名来替换通配符模式。在 C-Shell 和 Tcsh 中, 与此相同的功能称为文件名替换; 在 Korn Shell 和 Bourne Shell 中, 这一功能称为文件名生成。请参阅通配符(wildcard)、通配(globbering)和花括号扩展(brace expansion)。[24]

Pdksh Bourne shell 家族成员, 一个自由的开放源代码版本的 Korn shell, 最初由 Eric Gisin 于 1987 年开发。Pdksh 是“public domain Korn shell”只取首字母的缩写词。Pdksh 程序的名称是 ksh。请参阅 Korn shell 和 Bourne shell 家族(Bourne shell family)。[11]

权限(permission) 同文件权限(file permission)。[25]

PID 同进程 ID(process ID)。一个唯一的数字, 由内核分配, 用来标识一个特定的进程。发音为“P-I-D”。请参阅进程(process)和进程表(process table)。[26]

管道(pipe)

(1)管道线中两个连续程序之间的连接, 在该连接中, 一个程序的输出用作第二个程序的输入。有时候称为匿名管道。可与命名管道(named pipe)比较。

(2)(动词)将数据从一个程序发送给另一个程序, 从而创建一个管道线。[15]

管道线(pipeline) 两个或多个程序连续处理数据的一种布局, 一个程序的输出成为下一个程序的输入。[15]

指针(pointer) 指针指 GUI 中一个小的、可移动的图像, 该图像指示指点设备(鼠标)当前指向的位置。指针的形状是可以改变的, 取决于正在做的内容以及在屏幕上的位置。[6]

波兰表示法(Polish Notation) 运算符放在操作数前面的算术表示法, 例如“+ 5 7”。

该名称是为了纪念著名的波兰数学家、逻辑学家、哲学家 Jan Lukasiewicz(1878-1956)。同前缀表示法(prefix notation)。请参阅逆波兰表示法(Reverse Polish Notation)。[8]

弹出(pop) 和栈相关,从栈的顶部(也就是最后一个写入的元素)检索数据元素,同时将元素从栈中移除。[8] [24]

弹出式菜单(pop-up menu) 指在 GUI 中,作为某些动作(通常是右键单击)的结果在非明显位置出现的菜单。[6]

port 作为动词(指移植),指改编软件,使为某一计算机系统设计的软件可以在另一个系统上运行,例如“Tammy 将 Foo 程序由 Linux 移植(port)到 Windows。”作为名词,指这样的软件,例如“Tammy 创建了 Windows 版本(port)的 Foo 程序。”[2]

POSIX 一个项目,在 IEEE(Institute of Electrical and Electronics Engineers, 电气和电子工程师学会)的赞助下于 20 世纪 90 年代初期启动,目的是标准化 Unix。发音为“pause-ix”。POSIX shell 的规范由 IEEE 标准 1003.2 描述。Bourne shell 家族中的大多数现代 shell 都遵循这一标准。但是 C-Shell 家族中的 shell 并不是都遵循这一标准。[11]

POSIX 选项(POSIX option) 同 UNIX 选项(UNIX option)。[26]

后缀表示法(postfix notation) 运算符放在操作数之后的算术表示法,例如“5 7 +”。也称为逆波兰表示法(Reverse Polish Notation)。可与中缀表示法(infix notation)和前缀表示法(prefix notation)比较。[8]

预定义字符类(predefined character class) 和正则表达式相关,指一个可以用来替代字符类中一组字符的名称。例如,预定义字符类[:digit:]可以用来替代 0-9。请参阅字符类(character class)。[20]

前缀表示法(prefix notation) 运算符放在操作数之前的算术表示法,例如“+ 5 7”。也称为波兰表示法(Polish notation)。可与中缀表示法(infix notation)和后缀表示法(postfix notation)比较。[8]

前一作业(previous job) 与作业控制相关,指次最近挂起的作业,或者如果没有挂起作业的话,指最近移动到后台的作业。请参阅作业(job)、作业控制(job control)和当前作业(current job)。[26]

主组(primary group) 与具体用户标识相关,作为系统口令文件中用户标识的组列举的那个组。该用户标识所属的其他所有组都称为辅组。可与辅组(supplementary group)比较。请参阅组(group)和文件模式(file mode)。[25]

显示(print) 在终端上显示信息。例如,显示当前工作目录的命令是 pwd: “print working directory”。[7]

可显示字符(printable character) 在字符编码系统中,指那些可以显示和打印的字符,即非控制字符。在 ASCII 码中,共有 96 个可显示字符:字母、数字、标点符号、空格和(从实际应用来讲)制表符。请参阅 ASCII 码(ASCII code)。[19]

优先级(priority) 和进程执行的调度相关,指示相对于其他进程,某个进程拥有多大的优先次序。进程的优先级与进程的 nice 值成反比。请参阅进程(process)和 nice 值(nice number)。[26]

proc 文件(proc file) 一种伪文件,用来访问内核中的信息。在少数几种特殊情况中,proc 文件可以用来改变内核中的数据。最初,创建这些文件是为了提供有关进程的信息,

因此命名为“proc”。请参阅伪文件(pseudo file)。[23]

进程(process) 指加载到内存中并准备运行的程序, 以及程序的数据和跟踪程序所需的信息。进程由内核控制, 这与作业不同, 作业由 shell 控制。请参阅作业(job)、进程 ID(process ID)和进程表(process table)。[6] [15] [26]

进程 ID(process ID) 一个唯一的数字, 由内核分配, 表示一个特定的进程。通常称为 PID(“P-I-D”)。请参阅进程(process)和进程表(process table)。可与作业 ID(job ID)比较。[26]

进程表(process table) 一个由内核维护的表, 用来了解系统中的所有进程。每个进程在进程表中对应一个条目, 由 PID(进程 ID)索引。表中的每个条目包含描述及管理一个特定进程所需的信息。请参阅进程(process)和进程 ID(process ID)。可与作业表(job table)比较。[26]

进程树(process tree) 一种树型层次结构的数据结构, 显示父进程和子进程之间的连接。整个系统的进程树的根是初始化进程。请参阅进程(process)、父进程(parent process)、子进程(child process)和初始化进程(init process)。[26]

程序(program) 一串指令, 当计算机执行这串指令时就执行一项任务。[2]

Athena 计划(Project Athena) 麻省理工学院、IBM 公司和 DEC 公司研究人员之间的协作项目, 成立于 1984 年。其目的是为麻省理工学院的学生创建第一个标准化、网络化、与硬件无关的图形操作环境。Athena 计划显著的成就就是创建了第一版的 X Window。[5]

提示(prompt) 一个由程序显示的短消息, 指示程序准备好接收键盘的输入。[4]

伪文件(pseudo file) 3 种 Unix 文件中的一种。伪文件用来访问服务, 通常由内核提供。因为伪文件不存储数据, 所以它们不需要磁盘空间。最重要的伪文件类型有特殊文件(special file)、命名管道(named pipe)和 proc 文件(proc file)。可与普通文件(ordinary file)和目录(directory)比较。另请参阅文件(file)。[23]

伪终端(pseudo terminal) 一种在 GUI 中打开终端或连接到远程 Unix 计算机时使用的仿真终端, 缩写为 PTY。请参阅终端(terminal)。可与虚拟控制台(virtual console)比较。[23]

伪设备(pseudo-device) 一种充当输入源或输出目标但并不对应于实际设备(或者是真实的, 或者是仿真的)的特殊文件。两个最有用的伪设备是 null 文件(null file)和 zero 文件(zero file)。请参阅特殊文件(special file)。[23]

PTY 伪终端(pseudo terminal)的缩写。[23]

下拉菜单(pull-down menu) 在 GUI 中, 当单击某个单词或图标时出现的菜单。[6]

穿孔卡片(punch card) 一种卡片, 由美国发明家 Herman Hollerith(1860-1929)发明, 通过分列穿孔存储数据。[18]

压入(push) 和栈相关, 指将数据元素存储到栈中。[8] [24]

Q

队列(queue) 一种每次存储或检索一个元素的数据结构, 在这种数据结构中, 元素按存储的顺序被检索。请参阅 FIFO(first-in first-out, 先进先出)和数据结构(data structure)。[23]

引用(quote) 在 shell 中, 表示特定的字符根据特定的规则从字面意义上解释。这通过将一或多个字符放在单引号(')或双引号(")中, 或者在单字符前面放一个反斜线(\)来实现。请参阅强引用(strong quote)和弱引用(weak quote)。[13]

R

范围(range) 和正则表达式相关, 指字符类中一组可以排序的字符清单。范围由字符组的第一个成员, 后面跟一个连字符, 再后跟字符组的最后一个成员构成。例如, 范围 **0-9** 指数字 **0** 至 **9**。请参阅字符类(**character class**)。[20]

原始模式(raw mode) 一种键入字符作为程序输入的线路规程, 用户按下键后键入的字符就立即发送给程序。可与规范模式(**canonical mode**)和 **cbreak** 模式(**cbreak mode**)比较。另请参阅线路规程(**line discipline**)。[21]

读权限(read permission) 一种文件权限, 允许读取文件或目录。可与写权限(**write permission**)和执行权限(**execute permission**)比较。另请参阅文件权限(**file permission**)。[25]

重新引导(reboot) 一个停止 Unix, 然后再启动计算机的过程, 有效地停止并重启 Unix。[6]

记录(record) 包含机器可读数据的文件中的一行数据。请参阅字段(**field**)和定界符(**delimiter**)。[17]

递归(recursive)

(1)指根据自身来定义的算法或程序(计算机科学)或函数(数学)。非正式地讲, 描述一个可以无穷扩展的名称或只取首字母的缩写词, 例如, 递归只取首字母的缩写词 GNU 代表 “GNU's not Unix”。[2]

(2)和 Unix 文件命令相关, 描述处理目录的整个子树的选项。这样的选项通常命名为 **-r** 或 **-R**。[24]

重定向(redirect) 重新定义标准输入的源或标准输出及标准错误的目标。[15]

regex 正则表达式(**regular expression**)的缩写。[20]

正则表达式(regular expression) 一种基于具体元字符和缩写的规范, 提供简单明地描述了描述字符模式的方法。缩写为 “**regex**”, 或者更简单地缩写为 “**re**”。[20]

常规文件(regular file) 同普通文件(**ordinary file**)。[23]

相对路径名(relative pathname) 从当前目录开始解释的路径名。[24]

可移动介质(removable media) 能够在系统运行过程中插入或移除的存储设备, 例如 CD、DVD、软盘、磁带、闪存和内存卡。可与固定介质(**fixed media**)比较。[23]

重复次数(repeat count) 当使用 **vi** 文本编辑器时, 在命令前键入的一个数字, 使命令自动重复指定的次数。例如, 命令 **dd** 删除当前行。命令 **10dd** 从当前行开始删除 10 行。[22]

重复运算符(repetition operator) 在正则表达式中, 用来同时匹配多个字符的元字符 (*****、**+**、**?**、**{**、**}**)。请参阅限定(**bound**)。[20]

调整大小(resize) 在 GUI 中, 改变窗口的大小。[6]

重新启动(restart) 同重新引导(**reboot**)。[6]

恢复(restore) 在 GUI 中, 在窗口最小化或最大化后, 使窗口重新回到原始大小和位置。[6]

回车(return) 一个在发送给终端时使光标移动到下一行开头的字符。在 ASCII 码中, 回车字符的值为 13(十进制)或 0D(十六进制)。请参阅换行(**linefeed**)和新行(**newline**)。[7]

返回值(return value) 当程序或函数调用另一个程序或函数时, 发送回调用程序的数据。

例如, **fork** 系统调用向子进程发送 0 返回值, 向父进程发送非 0 返回值(子进程的进程 ID)。[26]

逆波兰表示法(Reverse Polish Notation) 运算符放在操作数之后的算术表示法, 例如“5 7 +”, 通常缩写为 RPN。采用这一名称是为了纪念著名的波兰数学家、逻辑学家和哲学家 Jan Lukasiewicz(1878-1956)。同后缀表示法(**postfix notation**)。[8]

修订控制系统(revision control system) 一种复杂的系统, 通常由软件开发人员使用, 用来管理大型程序或文档的开发。同源代码控制系统(**source control system**)。更专业的术语叫做版本控制系统(**version control system**)。[17]

右键(right button) 在鼠标或指点设备上, 当鼠标在右手时, 位于最右边的那个按钮。请参阅**左键(left button)**和**中间键(middle button)**。[6]

右键单击(right-click) 当使用鼠标或其他指点设备时, 按下右键。[6]

根(root)

(1) 一个特殊的用户标识, 为用户提供特殊的权限和极大的权利。为了维持适当的安全性, **root** 的口令是保密的, 只有系统管理员知道。以 **root** 登录的用户称为超级用户。[4]

(2) 在 Unix 文件系统中, 指主目录。同**根目录(root directory)**。根目录是文件系统中所有其他目录的父目录(直接或间接)。[23]

(3) 在树(数据结构)中, 形成树的主节点。请参阅**树(tree)**。[9]

根目录(root directory) Unix 文件系统的主目录。根目录是其他所有目录的直接或间接父目录。请参阅**根(root)**和**文件系统(filesystem)**。[23]

根文件系统(root filesystem) 存储在引导设备上的文件系统, 包含启动 Unix 所需的全部程序和数据文件, 以及系统出现问题时系统管理员所需的工具。[23]

路由器(router) 一种特殊用途的计算机, 将数据从一个网络中继到另一个网络。[3]

RPN 逆波兰表示法(Reverse Polish Notation)的缩写。[8]

RTFM

(1) 发音为 4 个单独的字母: “R-T-F-M”。在 Unix 文化中, RTFM 用作动词, 表示一种特定思想, 即在请求帮助之前, 应该首先在联机手册中查找信息。例如: “您可以帮助我了解 **sort** 命令吗? 我已经 RTFM 了它, 但是我还是不知道如何使用。”最初, RTFM 是 “Read the fuckin’ manual” 的缩写。现在, RTFM 是一个合法的单词, 拥有自己的含义, 并且是英语语言中没有元音的最长动词。[9]

(2) 从更为普遍的意义讲, RTFM 用来表示这样一种思想——在请求帮助之前, 自己应该阅读适当的文件资料, 或者在 Web 和 Usenet 上搜索相关的信息, 尝试着自己解决问题。[9]

运行(run) 遵循程序中包含的指令。同**执行(execute)**。[2]

运行级别(runlevel) 指一组 Unix 可以运行的模式, 确定提供哪些基本服务。从技术上讲, 即一种允许指定进程组存在的系统软件配置。请参阅**初始化进程(init process)**。[6]

运行时级别(runtime level) 同**运行级别(runlevel)**。[6]

S

调度器(scheduler) 一种由内核提供的服务, 用来跟踪等待处理器时间的进程, 从而

确定接下来执行哪个进程。[6] [26]

屏幕编辑器(screen editor) 同面向屏幕的编辑器(screen-oriented editor)。[22]

面向屏幕的编辑器(screen-oriented editor) 一种允许在屏幕的任何位置输入、显示及管理数据,且不需要使用要求行号的命令的文本编辑器。同屏幕编辑器(screen editor)。可与面向行的编辑器(line-oriented editor)比较。[22]

滚动(scroll) 在终端的屏幕上移动行——通常是向上或向下,从而为新行留出空间。[7]

搜索路径(search path) 在 shell 中,当需要查看一个必须执行的程序时,shell 查看程序的目录列表。[13]

辅助提示(secondary prompt) 一个用来表示命令在新行上继续显示的特殊 shell 提示。请参阅 shell 提示(shell prompt)。[19]

句子(sentence) 当使用 vi 文本编辑器时,以一个句点、逗号、问号或感叹号结尾,后面至少跟两个空格或一个新行字符的一串字符。有一些 vi 命令在移动光标或修改文本时作用于句子。请参阅单词(word)和段落(paragraph)。[22]

服务器(server)

(1)一个提供某类服务的程序,通常是通过网络。请求此类服务的程序称为客户程序。[3]

(2)运行服务器程序的计算机。[3]

(3)由微内核用来执行具体任务的程序。[2]

设置(set)

(1)在 shell 中,指创建一个变量,并且还有可能给这个变量赋一个值。[12]

(2)在 Bourne shell 家族(Bash、Korn shell)中,指打开一个选项。请参阅复位(unset)。[12]

setuid 通常缩写为 suid(发音为“S-U-I-D”),代表“set userid(设置用户标识)”。一种只对包含可执行程序的文件使用的特殊文件权限。当设置 setuid 时,程序以文件属主的权限执行程序,而不管哪个用户标识运行该程序。setuid 通常用来使常规的用户标识运行要求拥有超级用户权限的 root 拥有的程序。[25]

影子文件(shadow file) 指系统文件/etc/shadow,该文件中包含编辑后的口令及相关的数,例如过期日期。请参阅口令文件(password file)。[11]

shell 通过充当命令处理器以及解释命令脚本,提供 Unix 基本界面的程序。[2] [11]

shell 选项(shell option) 在 Bourne shell 家族(Bash、Korn shell)中使用,指为了控制 shell 行为的特定方面而充当 off/on 开关的设置。在 C-Shell 家族(C-Shell、Tcsh)中,不使用 shell 选项。实际上,C-Shell 家族中的 shell 的行为受 shell 变量的设置控制。[12]

shell 提示(shell prompt) 由 shell 显示的一个或多个字符,指示其已经准备好接受新命令。请参阅辅助提示(secondary prompt)。[4] [13]

shell 脚本(shell script) 一串存储在文件中可以由 shell 执行的命令。大多数 shell 都有专门设计的编程命令,可以在 shell 脚本中使用。[11] [12]

shell 变量(shell variable) 在 shell 中不属于环境变量的局部变量,因此无法被子进程访问。请参阅环境变量(environment variable)、局部变量(local variable)和全局变量(global variable)。[12]

快捷键(shortcut key) 在 GUI 中,允许不必打开菜单选择相应项而执行特定动作的键或键组合。[6]

关机(shutdown) 停止 Unix 并关闭计算机的过程。[6]

信号(signal) 一种进程间通信, 在这类通信中, 一个数字形式的简单消息发送给进程, 从而使进程知道发生了某些事情。信号的识别由进程完成, 然后进程再采取相应的动作。当进程完成这一过程后, 我们就称进程捕获了信号。请参阅进程间通信(interprocess communication)和捕获(trap)。[26]

单用户模式(single-user mode) 运行级别 1。一种只能由超级用户登录的运行级别, 通常用来执行系统维护或修复。[6]

Slackware 第一个成功的 Linux 发行版, 由 Patrick Volkerding 于 1993 年 7 月发行。名称“slack”是一个异想天开的选择, 取自 SubGenuis 教堂, 一个拙劣的模仿宗教。slack 指实现个人目的的愉快和满足感。

软链接(soft link) 符号链接(symbolic link)的同义词。用来区分常规链接(硬链接)和符号链接(软链接)。[25]

软件(software) 所有类型的计算机程序。[2]

源(source) 源代码(source code)的同义词。[2]

源代码(source code) 用计算机语言编写的程序, 知识渊博的用户或程序员可以阅读。在将源程序转换成可执行程序时, 源程序必须翻译成机器语言。非正式地讲, 源代码通常简称为“源”。[2]

源代码控制系统(source control system) 一种复杂的系统, 通常由软件开发人员使用, 用来管理大型程序或文档的开发。同修订控制系统(revision control system)。更通用的术语为版本控制系统(version control system)。[17]

特殊文件(special file) 一种伪文件, 提供物理设备的内部表示, 也称为设备文件(device file)。请参阅伪文件(pseudo file)。[23]

挤压(squeeze) 和修改文本的程序(例如 tr)相关, 当执行转换操作时, 将多个相邻的相同字符视为一个单独的字符。例如, 将多个连续空格替换为一个空格。请参阅转换(translate)。[19]

栈(stack) 一种数据结构, 在这种数据结构中, 同时只能存储或检索一个元素, 也就是在任何时候, 下一个检索的元素是最后一个存储的元素。请参阅 LIFO(“last-in first-out, 后进先出”)和数据结构(data structure)。[8] [24]

标准错误(standard error) 程序在写入错误消息时的默认目标。缩写为 stderr。当用户登录时, shell 会自动地将标准错误设置为显示器。因此, 默认情况下, 许多程序将错误消息写到显示器上。请参阅标准输入(standard input)、标准输出(standard output)和标准 I/O(standard I/O)。[15]

标准 I/O(standard I/O) 一个集合名词, 指标准输入(standard input)、标准输出(standard output)和标准错误(standard error), 是“standard input/output, 标准输入/输出”的缩写。[15]

标准输入(standard input) 程序的默认输入源, 缩写为 stdin。当用户登录时, shell 会自动地将标准输入设置为键盘。因此, 默认情况下, 许多程序从键盘读取输入。请参阅标准输出(standard output)、标准错误(standard error)和标准 I/O(standard I/O)。[15]

标准选项(standard option) 同 UNIX 选项(UNIX option)。[26]

标准输出(standard output) 程序写结果的默认目标, 缩写为 **stdout**。当用户登录时, shell 会自动地将标准输出设置为显示器。因此, 默认情况下, 许多程序将输出写到显示器上。请参阅标准输入(**standard input**)、标准错误(**standard error**)和标准 I/O(**standard I/O**)。[15]

状态(state) 进程的当前状态。在任何时候, 进程处于 3 种可能状态中的一个: 正在前台运行; 正在后台运行; 挂起(暂停), 等待信号恢复执行。请参阅前台进程(**foreground process**)、后台进程(**background process**)和挂起(**suspend**)。[26]

静态数据(static data) 在 Unix 文件系统中, 指如果没有系统管理员的干预就不会改变的数据。可与可变数据(**variable data**)比较。[23]

stderr 标准错误(**standard error**)的缩写。[15]

stdin 标准输入(**standard input**)的缩写。[15]

stdout 标准输出(**standard output**)的缩写。[15]

停止(stop) 使进程临时暂停, 通常通过按 **^Z** 键实现。一旦进程停止, 那么该进程就等待信号恢复执行。同挂起(**suspend**)。可与杀死(**kill**)相比较。请参阅进程(**process**)和作业控制(**job control**)。[26]

流(stream) 或者由程序读取(作为输入流(**input stream**)), 或者由程序写入(作为输出流(**output stream**))的数据。[19]

串(string) 一串可显示的字符。同字符串(**character string**)。请参阅可显示字符(**printable character**)。[19]

强引用(strong quote) 当使用 shell 时, 和单引号(')是同义词。在单引号中, 字符没有特殊的含义。可与弱引用(**weak quote**)比较。[13]

子目录(subdirectory) 也称为 child directory(子目录)。指位于另一个目录中的目录。除根目录之外, 可以认为所有的目录都是子目录。包含子目录的目录称为父目录。请参阅目录(**directory**)。[23]

子 shell(subshell) 由另一个 shell 启动的 shell。[15] [26]

suid setuid 的缩写, 发音为 “S-U-I-D”。[25]

超块(superblock) 指 Unix 文件系统中, 一个用来存放文件系统本身的关键信息的特殊数据区域。[24]

超级用户(superuser) 指一个用户, 通常是系统管理员, 使用用户标识 **root** 登录, 拥有特殊的权利。请参阅根(**root**)。[4]

辅组(supplementary group) 和具体用户标识相关, 指除了用户标识所属的主组之外, 该用户标识所属的其他组。可与主组(**primary group**)比较。请参阅组(**group**)和文件模式(**file mode**)。[25]

挂起(suspend) 临时暂停进程, 通常通过按下 **^Z** 键实现。一旦进程被挂起, 它就会等待信号恢复执行。同停止(**stop**)。可与杀死(**kill**)比较。另请参阅进程(**process**)和作业控制(**job control**)。[26]

交换文件(swap file) 当使用 **vi** 文本编辑器时, 在正编辑的文件所在目录中自动保存的编辑缓冲区的一个副本。如果工作任务突然终止——例如系统崩溃——那么交换文件可以用来恢复数据。[22]

开关(switch)

(1)当输入 Unix 命令时: 选项的另一个名称。[10]

(2)当使用 **vi** 文本编辑器时：一种或者为开(on)，或者为关(off)的选项。可与变量(variable)比较。请参阅选项(option)。[22]

符号链接(symbolic link) 指文件系统中的一种链接类型，从字面意义上讲，也就是另一个文件的路径名。符号链接有时候也称为软链接，以区分常规链接(硬链接)。符号链接通常称为 **symlink**。可与链接(link)比较。[25]

symlink 符号链接(symbolic link)的缩写。[25]

语法(syntax) 如何输入命令的形式描述。[10]

sysadmin 系统管理员(system administrator)的同义词。[4]

系统管理员(system administrator) 管理及组织 Unix 系统的人，同 **sysadmin**。[4]

系统调用(system call) 一种由进程使用，调用内核提供的服务的功能。[12]

系统维护模式(system maintenance mode) 单用户模式(single-user mode)的废弃术语。[6]

系统管理员(system manager) 系统管理员(system administrator)的废弃术语。

System V 一种 Unix 版本，由 AT&T 公司开发，于 1983 年发布。在 20 世纪 80 年代，System V 是 Unix 两大主要分支之一，另一个主要分支是 **BSD**。[2]

T

制表位(tab stop) 在电传打字机上，一个用来设置按<Tab>键时回车停止位置的机械标记。[18]

任务(task) 在 GUI 中，在窗口中正在运行的程序。[6]

任务切换(task switching) 在 GUI 中，指将焦点从一个窗口改变到另一个窗口，通常使用键组合在当前正运行任务的列表之间循环。对于大多数 GUI 来说，任务切换键是<Alt-Tab>(在列表中向前移动)和<Alt-Shift-Tab>(在列表中向后移动)。[6]

任务栏(taskbar) 在 GUI 中，指一个通常位于屏幕底部的水平栏，在任务栏上包含有当前活跃的每个窗口的表示。窗口的表示通常是图标，还可能有一些文本。在 Gnome 中，任务栏的功能通常由“Window List”提供。[6]

TCO 总拥有成本(total cost of ownership)的缩写。[5]

Tcsh C-Shell 家族成员之一，最初开发于 20 世纪 70 年代末，由 Carnegie-Mellon 大学的 Ken Greer 作为 C-Shell 的一个完全免费的版本开发。发音为“tee-see-shell”。Tcsh 作为一个功能强大、向后兼容并替换传统 C-Shell 的 shell 被广泛使用。Tcsh 程序的名称是 **tcsh** 或 **csh**。请参阅 C-Shell 家族(C-Shell family)。[11]

Termcap 一个由一个大文件构成的数据库，包含有所有不同类型终端的技术描述信息。在现代系统中，Termcap 已经被 **Terminfo** 取代。[7]

终端(terminal) 用来通过键盘、显示器及鼠标访问 Unix 系统的硬件。Unix 终端可以是作为终端设计的机器，也可以是运行程序充当(仿真)终端的计算机。[3]

终端驱动程序(terminal driver) 充当终端驱动的程序。[21]

终端室(terminal room) 在 20 世纪 70 年代和 80 年代，一个用来存放大量连接到主机计算机的终端的房间。为了使用 Unix 系统，用户需要到终端室去，等待空闲的终端。[3]

终端服务器(terminal server) 一种特殊用途的计算机,充当将终端连接到主机计算机的开关。[3]

终止(terminate) 和进程相关,指停止进程的运行。同死亡(die)。[26]

Terminfo 一个由一组文件构成的数据库,包含所有不同类型终端的技术描述信息。在现代系统中,Terminfo 已经取代了较旧的数据库 Termcap。[7]

测试(test) 当使用 **find** 程序时,一个用来定义文件搜索过程中所使用标准的规范。例如,测试-type f 告诉 **find** 只搜索普通文件。请参阅动作(action)和运算符(operator)。[25]

Texinfo GNU 项目的官方文档资料系统。Texinfo 提供了一组复杂的工具,使用一个信息文件以不同格式生成输出: Info 格式、纯文本、HTML、DVI、PDF、XML 和 Docbook。最常见的发音为“Tekinfo”。[9]

文本(text) 由字符构成的数据,这些字符包含字母、数字、标点符号等。[3]

文本编辑器(text editor) 用来创建和修改文本文件的程序,通常非正式地称为编辑器。[22]

文本文件(text file) 只包含可显示字符的文件,而且每行的末尾还有一个新行字符。Unix 过滤器就是设计用来处理文本文件的。有时候称文本文件为 ASCII 文件。可与二进制文件(binary file)比较。请参阅可显示字符(printable character)。[19]

基于文本的终端(text-based terminal) 一种只显示字符(文本),即字母、数字、标点符号等的终端。同字符终端(character terminal)。[3]

时间片(time slice) 一个非常短的时间间隔,在这段间隔中,允许一个特定的进程使用处理器。典型的时间片通常是 10 毫秒(千分之十秒)。请参阅 CPU 时间(CPU time)。[6] [26]

分时系统(time-sharing system) 多用户系统的旧称。指操作系统可以同时支持多名用户。[3]

标题栏(title bar) 在 GUI 中,窗口顶端显示窗口中正在运行程序的名称的水平区域。[6]

顶(top) 在栈中,最近被压入(写入)到栈中的数据元素的位置。[8] [24]

Top 节点(Top Node) 指 Info 系统中树的根。通常,Top 节点包含所讨论主题的摘要,以及一个显示该文件涉及话题的菜单。[9]

顶级目录(top-level directory) 根目录的任何子目录。根目录和顶级目录形成 Unix 文件系统的主干。请参阅根目录(root directory)和文件系统(filesystem)。[23]

总拥有成本(total cost of ownership) 一个商业术语,通常缩写为 TCO。指在机器或系统生命周期中拥有和使用它的开销总费用估计。为了估计计算机的 TCO,必须考虑计算机硬件、软件、升级、维护、技术支持以及培训的开销。通常,商业 PC 的 TCO 是其购买价格的 3 到 4 倍。[5]

全序(total ordering) 当对一组元素排序时,排序组内全部成员的二元关系。可与偏序(partial ordering)相比较。[19]

转换(translate) 和修改文本的程序(例如 tr)相关,将字符的每个实例改变为一个或多个指定的字符。请参阅挤压(squeeze)。[19]

封闭(trap) 针对于正在执行的程序,消除其对具体信号的注意和反应,特别是可能终止程序或产生其他影响的信号。[7]

树(tree) 一种由一组节点、叶子和分支形成的数据结构,按照任意两个节点之间至多

一个分支的方式组织。请参阅数据结构(**data structure**)、节点(**node**)、叶子(**leaf**)、分支(**branch**)和根(**root**)。[9]

三击(triple-click) 当使用鼠标或其他指点设备时,快速连续地按键 3 次。[6]

U

UI Unix 国际(**Unix International**)的缩写。[5]

Unix

(1)泛指任何在提供用户和编程服务方面满足一般公认的“类 Unix”标准的操作系统。

(2)描述一个全球范围内的文化,该文化基于 Unix 操作系统,包括界面、shell、程序、语言、习俗和标准。[2]

UNIX 最初由 AT&T 公司开发的一个操作系统产品及关联软件家族。可与 **Unix** 比较。[2]

Unix 国际(Unix International) 一个由 AT&T 公司、Sun 公司和其他几家小公司于 1989 年 12 月成立的组织,和开放软件基金会的作用相同。缩写为 **UI**。[5]

Unix 手册(Unix manual) 同联机手册(**online manual**)。[9]

UNIX 选项(UNIX option) 与 **ps**(process status, 进程状态)命令相关,这些选项派生于 20 世纪 80 年代 AT&T UNIX 版的 **ps** 选项。UNIX 选项以一个破折号开头。可与 **BSD 选项(BSD option)** 比较。也称为 **POSIX 选项(POSIX option)** 和 **标准选项(standard option)**。[26]

向下还原按钮(unmaximize button) 指 GUI 中的一个小方框,通常位于窗口的右上角,当单击这个小方框时,将恢复先前最大化的窗口。[6]

卸载(unmount) (动词)断开设备文件系统与主 Unix 文件系统的连接,禁止对它的访问。卸载文件系统时,要使用 **umount** 命令。请参阅挂载(**mount**)和文件系统(**filesystem**)。[23]

无名缓冲区(unnamed buffer) 当使用 **vi** 文本编辑器时,一个包含最后一次删除内容副本的存储区域。请参阅编号缓冲区(**numbered buffer**)。[22]

复位(unset)

(1)在 shell 中,指删除一个变量。[12]

(2)在 Bourne shell 家族(Bash、Korn shell)中,指关闭一个选项。请参阅设置(**set**)。[12]

大写字母(uppercase) 描述大写字母“A”至“Z”。[4]

Usenet 一个全球范围的讨论组。[3]

用户(user) 以某种方式使用 Unix 的人。Unix 不知道用户,它只知道用户标识。[4]

用户掩码(user mask) 一个由 3 个八进制数字构成的数,表示新创建的文件所具有的文件权限。[25]

用户名补全(user name completion) 同用户标识补全(**userid completion**)。[13]

用户标识(userid) 一个在 Unix 系统中注册的名称,用来标识一个具体的账户。发音为“user-eye-dee”。[4]

用户标识补全(userid completion) 一种自动补全,当单词以~(波浪号)字符开头时,补全局部键入的用户标识。Bash、C-Shell 和 Tcsh 提供有用户标识补全。请参阅自动补全(**autocompletion**)。[13]

实用工具(utility) 指随 Unix/Linux 操作系统一起发行的数百个程序。[2]

V

值(value) 存储在变量中的数据。[12]

变量(variable)

(1)一个由名称知道的数量,表示一个值。[12]

(2)当使用 **vi** 文本编辑器时,指一个包含值的选项。可与开关(**switch**)比较。另请参阅选项(**option**)。[22]

变量补全(variable completion) 一种自动补全,当单词以\$(美元符号)字符开头时,补全部分键入的变量名。Bash 和 Tcsh 提供有变量补全。请参阅自动补全(**autocompletion**)。[13]

可变数据(variable data) 在 Unix 文件系统中,随着时间变化而变化的数据,例如日志文件。可与静态数据(**static data**)比较。[23]

版本控制系统(version control system) 一种复杂系统的通用名称,通常由软件开发人员和工程师使用,管理大型程序、文档、蓝图等的开发。当由程序员使用时,通常称为源代码控制系统(**source control system**)或修订控制系统(**revision control system**)。[17]

vi 一种功能强大、基于屏幕的文本编辑器,每个 Unix 系统都提供该编辑器。**vi** 编辑器是事实上的标准 Unix 文本编辑器。名称 **vi** 的发音是两个单独的字母“vee-eye”。请参阅 **Vim**。[22]

vi 模式(vi mode) 在 shell 中,一种命令行编辑时使用的模式,在这种模式中,编辑命令与 **vi** 文本编辑器中使用的命令相同。请参阅命令行编辑(**command line editing**)和 Emacs 模式(**Emacs mode**)。[13]

Vim 一种功能非常强大,向后兼容并取代 **vi** 的文本编辑器。在许多 Unix 和 Linux 系统上,默认情况下,Vim 已经取代了 **vi**。请参阅 **vi**。[22]

虚拟控制台(virtual console) 在同一时间运行的若干个终端仿真程序中的一个,每个终端仿真程序都支持一个独立的工作会话。在 Linux 中,最常见的默认配置为用户提供 7 个虚拟控制台:1~6 号控制台是全屏的,使用 CLI 的基于文本的终端;7 号控制台是图形终端,运行 GUI。在这样的系统中,桌面环境(例如 KDE 和 Gnome)运行在虚拟控制台 7 中。请参阅终端(**terminal**)。可与伪终端(**pseudo terminal**)比较。[6]

虚拟文件系统(virtual filesystem) 一种不管数据的存储和生成方式,为程序访问数据提供统一方式的 API(application program interface,应用程序界面)。虚拟文件系统使得将单独、异构的设备文件系统组织成一个大的 Unix 文件系统成为可能。缩写为 VFS。[23]

访问(visit) 指在 Info 系统中,查看特定节点的内容。[9]

VT100 在所有时间都最流行的 Unix 终端,于 1978 年由 DEC 公司开始推行。VT100 非常流行,以至于成为一个永久的标准。即使在现在,许多终端仿真程序所使用的规范都基于 VT100。[3]

W

等待(wait) 在进程分叉创建子进程后, 进程暂停直至子进程结束运行的过程。请参阅分叉(**fork**)、**exec** 和退出(**exit**)。[26]

弱引用(weak quote) 当使用 shell 时, 是双引号(")的同义词。在双引号中, 只有 3 个元字符保留它们的特殊含义, 这 3 个元字符是 \$(美元符号)、`(反引号)和 \ (反斜线)。可与强引用(**strong quote**)比较。[13]

Web 服务器(Web server) 一种存储 Web 页面并使 Web 页面通过网络(通常指 Internet)可用的计算机。[3]

空白符(whitespace)

(1)对于 shell 来说, 指一个或多个连续的空格或制表符。

(2)对于一些程序来说, 指一个或多个连续的空格、制表符或新行字符。[10]

通配符(wildcard) 当指定文件名时(通常在 Unix 命令中), 指用来创建可以匹配多个文件的模式的元字符。请参阅通配(**globbing**)和路径名扩展(**pathname expansion**)。[24]

窗口(window) 当使用 GUI 时, 指屏幕的一个有界区域, 通常是一个矩形。[5]

窗口管理器(window manager) 在 GUI 中, 用来控制图形元素(窗口、按钮、滚动条、图标等)外观和特性的程序。[5]

窗口操作菜单(window operation menu) 在 GUI 中, 当使用窗口时, 指窗口本身中包含一串动作(例如 Move、Resize、Minimize、Maximize 和 Close)的一个下拉菜单。为了显示窗口操作菜单, 需要单击窗口左上角的小图标(位于标题栏的左边缘)。[6]

word

(1)(字)特定处理器组织及管理位的基本单位, 许多现代的处理使用 32 位或 64 位的字。[21]

(2)(单词)当使用正则表达式时: 一个自我包含, 由字母、数字或下划线字符构成的连续字符序列。[20]

(3)(单词)当使用 vi 文本编辑时: 一个由字母、数字或下划线字符构成的字符串。当移动光标或修改文本时, 有多个 vi 命令作用于单词。[22]

工作目录(working directory) 也称为当前目录(**current directory**)。当输入 Unix 命令时, Unix 命令使用的默认目录。工作目录通过 **cd**(change directory, 改变目录)命令设置, 显示工作目录名称的命令则是 **pwd**(print working directory, 显示工作目录)。[24]

工作空间(workspace) 在 Gnome 桌面环境中, 指桌面(**desktop**)。[6]

写权限(write permission) 一种文件权限。对于文件来说, 该权限允许写入文件。对于目录来说, 该权限允许在目录中进行创建、移动、复制或删除操作。可与读权限(**read permission**)和执行权限(**execute permission**)比较。另请参阅文件权限(**file permission**)。[25]

X

X 同 **X Window**。[5]

X 终端(X terminal) 指任何设计使用 X Window 系统的图形终端。现在, X 终端标准是图形终端仿真的基础, 就如同 VT100 是字符终端仿真的基础一样。[3]

X Window 一种广泛使用的系统, 用来支持图形用户界面(GUI)。该术语的正确用法是单数“X Window”, 而不是复数“X Windows”。X Window 系统通常简称为“X”。[5]

Y

接出(yank) (动词)当使用 **vi** 文本编辑器时, 将文件复制到无名缓冲区而不删除文本。请参阅无名缓冲区(**unnamed buffer**)。[22]

Z

zero 文件(zero file) 指伪文件/**/dev/zero**。当作为输出目标使用时, zero 文件将抛弃所有的输入。当作为输入源使用时, zero 文件总是返回一个 null 字符。zero 文件是两个位桶(**bit bucket**)中的一个, 另一个就是 **null 文件(null file)**。请参阅伪文件(**pseudo-file**)和 **null 字符(null character)**。[23]

0 个或多个(zero or more) 指示可以使用 1 个或多个指定对象, 或者完全省略。例如, 某命令的语法允许指定 0 个或多个文件名。这意味着可以指定 1 个或多个文件名, 也可以将文件名完全省略。可与一个或多个(**one or more**)比较。[10]

僵尸进程(zombie) 一种已经死亡, 但是父进程还没有将它标识为消失的子进程。如果子进程是孤儿(没有父进程), 那么该子进程将一直是孤儿, 直至系统执行某些动作使该子进程消失。请参阅死亡(**die**)和孤儿(**orphan**)。[26]

Zsh

Bourne shell 家族成员之一, 一种功能非常强大、非常复杂的 shell, 最初由普林斯顿大学的学生 Paul Falstad 于 1990 年开发。Zsh 的发音为“zee-shell”。Zsh 程序的名称是 **zsh**。请参阅 Bourne shell 家族(**Bourne shell family**)。[11]